INCREMENTAL PARSER CONSTRUCTION WITH METAOBJECTS

Kai Koskimies and Juha Vihavainen

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1992-5

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1992-5, NOVEMBER 1992

INCREMENTAL PARSER CONSTRUCTION WITH METAOBJECTS

Kai Koskimies and Juha Vihavainen

University of Tampere
Department of Computer Science
P.O. Box 607
SF-33101 Tampere, Finland

ISBN 951-44-3249-5
ISSN 0783-6910

Incremental Parser Construction with Metaobjects

Kai Koskimies
Department of Computer Science, University of Tampere
email: koskimie@cs.uta.fi

Juha Vihavainen
Department of Computer Science, University of Helsinki
email: vihavain@cs.Helsinki.fi

Abstract
The construction of an object-oriented recursive descent parser is studied. A grammar is modelled by representing each nonterminal symbol as a class. To support interactive, incremental parser construction, it is required that a modification in the definition of a nonterminal has minimal effects on the classes of other nonterminal symbols. An object-oriented parsing method based on metaobjects and lazy recursive descent technique is developed. It is shown that Eiffel allows a pseudo-incremental solution in which the changes propagate only to the next superclass level, while C++ allows fully incremental solution.

1 Introduction

An important principle in modern software construction is *incrementality*. Basically, this means that software is composed of separately compiled units, and that a system and its behavior can be extended by introducing new such units, without changing the existing parts of the system. In object-oriented software construction such a unit is a class. Incrementality makes the software development process more manageable: a system can be constructed and tested through stepwise extensions, starting from a skeletal core system and gradually "plugging in" units implementing new functionalities. Incrementality supports software continuity [Mey88], because changes in the requirements of a particular functionality are likely to affect only a single unit in the software. In automatic program synthesis incrementality becomes an efficiency issue: if software is generated automatically on the basis of changing and increasing knowledge of the application, incrementality allows the generator to avoid the regeneration (and recompilation) of large parts of the system after some modification of the application's requirements.

The purpose of this paper is to study the construction of a top-down parser from the viewpoint of object-oriented software construction. Our special requirement is that a language is modelled by representing each nonterminal symbol as a class, and that changes in the definition of a particular nonterminal have minimal effect on the classes of other nonterminal symbols. This requirement is motivated by incremental parser generation: a parser can be generated and compiled piecewise, one nonterminal at a time, and existing nonterminal definitions can be changed without regenerating and recompiling the whole parser. This is essential in an interactive language implementation system: the user should be able to edit nonterminal definitions without being forced to wait an unreasonable amount of time after each editing action. The problem was actually encountered in the design of the TaLE system [JKP91]; this is a specialized Eiffel based editor supporting the development of language implementations.

We apply the idea of metaobjects [Fer89] in parsing: each nonterminal is represented at run time by a metaobject whose task is to create an instance of the proper nonterminal class on the basis of the input stream. We use an object-oriented interpretation of context-free grammars (see next section) in which alternative forms of a nonterminal are viewed as subclasses. Hence, if a metaobject represents a nonterminal having subclasses, it has to know which subclass should be instantiated; the metaobject has to collect the necessary parsing information when it is created. This implies that (at least some) parsing information must be collected at run time, in a "lazy" fashion. For this purpose we employ the so-called lazy recursive descent scheme [Kos90a]. We study concrete solutions in terms of two different strongly typed languages: a pure object-oriented language (Eiffel), and a hybrid language (C++). We show that in Eiffel only a partially incremental solution is possible, while C++ allows full incrementality.

During the last few years, object-oriented language implementation techniques have been advocated by several authors (e.g. [Lie87], [CNS87], [WuW92]). In language implementation systems, object-orientation has been successfully combined with the specification formalisms in various ways, in particular with attribute grammars (for a survey, see [Kos91]). A notable system in this direction is Mjølner/Orm [Mag90] in which incrementality is achieved by applying a generator of an interactive, incremental programming environment to the metalanguage of the system itself: in this way the user can edit a language specification incrementally, and see the results immediately in the behavior of the language environment. The system is driven by an interpreter reading the internal, incrementally updated data structures representing the language specification.

Our approach is less ambitious: we develop techniques for constructing a parser incrementally using directly a common object-oriented programming language rather than an interpreter-based system. Our techniques can be applied in hand-written language processors as well as in automated language implementation, in the same way as the traditional recursive descent parsing method can be used in both hand-written and automatically generated language processors. The efficiency of the resulting parser is expected to be comparable to the traditional case.

Incremental, non-object-oriented language implementation techniques have been developed by the ASF+DSF group [Kli91]. In particular, the group has developed an incremental LR parsing method [HKR89]. This method is based on a lazy approach in which LR parsing tables are constructed during parsing. The notable advantage of their method is generality: in fact their method accepts any context-free grammar (due to the use of a pseudo-parallel variant of the LR method). Another work concerning the incremental construction of LR parsers is [Hor90] in which an incremental variant of LALR(1) parser generation is presented. Both these works employ table-driven LR techniques which make them essentially different from ours. Since our method is based on LL(1) parsing, the acceptable grammars are more restricted, but on the other hand our method leads to simpler and faster parsers. Incremental LL parser construction has been studied also in [Gyi88]; a main difference with their work is that they use table-driven parsers rather than hard-coded recursive descent. Consequently, in their system incrementality concerns parsing tables rather than programs. Incrementally constructed, but non-object-oriented and non-deterministic top-down parsers have been used already in [HeR75] and in [Gro84].

We proceed as follows. Section 2 explains concepts our work is based on; in particular, we discuss the object-oriented forms of context-free grammars and the lazy approach to recursive descent parsing. We also explain our short-hand notation for defining class texts. In Section 3 we define the problem more specifically and discuss the solution on a general, language independent level. Sections 4 and 5 present the Eiffel and C++ solutions, respectively. Finally, in Section 6 we compare the two solutions and discuss the overall benefits of the method. We assume that the reader is familiar with object-oriented programming, and roughly understands Eiffel and C++. We use version 3 of Eiffel [Mey92], and Borland C++ [Bor91].

2 Background

Object-oriented context-free grammars

Variations of object-oriented context-free grammars have been used in many contexts, with different names and purposes (e.g. [Ten88], [LMN88], [Kos88]; for a survey, see [Kos91]). Essentially, an object-oriented context-free grammar (OO-CFG) is a restricted form of CFG where each alternative structural form of a nonterminal is named with another nonterminal symbol. For example, production rules

```
Statement -> Identifier ":=" Expression | "if" Expression "then" Statement "end"
```

would be forbidden in an OO-CFG, because Statement has two unnamed alternatives; the grammar has to be rewritten in the form

```
Statement -> Assignment | IfStatement
Assignment -> Identifier ":=" Expression
IfStatement -> "if" Expression "then" Statement "end"
```

The reasoning behind this requirement is that nonterminals are regarded as classes and alternatives are viewed as subclasses; the requirement follows simply from the fact that each class must have a name. In the example grammar, Assignment and IfStatement are regarded as subclasses for Statement. As demonstrated by this example, an OO-CFG is usually a natural way to model a language, and language designers tend to write syntactic specifications in this form, even without a conscious goal of object-orientation.

An OO-CFG divides nonterminals (we will continue to speak of "nonterminals" instead of "classes" in this context, to avoid confusion with the classes used to implement parsing) into two categories: those having a single structural specification consisting of a sequence of nonterminal and terminal symbols (e.g. Assignment, IfStatement), and those having a specified set of "subclass" nonterminals (e.g. Statement); we call the former *structural* nonterminals and the latter *collective* nonterminals, following the terminology in [JKP91]. Note that the production rules are interpreted in a particular way in an OO-CFG: for a structural nonterminal, the production arrow "->" is understood as "consists of the following components", while for a collective nonterminal this arrow is understood as "has the following subclasses". In the latter case, the syntactic effect is a consequence of inclusion polymorphism: if B is a subclass of A, then any context requiring an A can also accept a B; hence an A can be replaced by B in a grammatical sense. To make the distinction between structural and collective nonterminals more explicit, we use in the sequel different rule symbols: "->" for a structural nonterminal and ">" for a collective nonterminal. If A is a collective nonterminal with rule A>...|B|..., we say that A is the *parent* nonterminal of B, and B is a *child* nonterminal of A. The transitive closures of these relations are called *ancestor* and *descendant* nonterminal, respectively.

An OO-CFG implies an internal form of the source text as a collection of objects created as instances of the nonterminals, the nonterminals being viewed as classes. A structural nonterminal has a special attribute for each nonterminal symbol occurring on the right-hand side of its rule; we call these attributes the *syntactic attributes* of the nonterminal. The value of a syntactic attribute is a reference to the object representing the corresponding component. A collective nonterminal needs no syntactic attributes: it is never instantiated as such, but only as a superclass layer of an instance of a structural nonterminal. The resulting object graph is close to a traditional syntax tree; however, this form is more abstract because typical unit productions (i.e. productions with a single nonterminal on the right-hand side) do not give rise to "nodes" (i.e. objects) in the tree structure. Terminal symbols are also ignored in this representation.

In principle OO-CFG could allow multiple inheritance, i.e. rules of the form A>B, C>B, but for various reasons practical systems often forbid this; we do the same here. Since we are mainly interested in parsing, we also forbid nonterminals that are syntactically useless, although such "abstract" nonterminals might be sensible in a semantic model of a language (as e.g. in [Hed89] and [JKP91]). Finally, since class hierarchies cannot be cyclic (i.e. a class cannot inherit itself), we require that an OO-CFG is acyclic with respect to the > relation: it is not allowed that $A>B_1>...>B_k>A$ for some A ($k\geq0$). These additional restrictions imposed by the object-oriented interpretation lead us to the following definition (this is a variation of the definition of strongly SI-structured context-free grammars given in [Kos91]):

An OO-CFG is a 6-tuple (S,C,T,P,H,Z), where S is the set of structural nonterminal symbols, C is the set of collective nonterminal symbols, T is the set of terminal symbols, P is the set of productions of the form N->w ($N\in S$, $w\in(S\cup C\cup T)^*$), H is the set of hierarchy relations of the form A > B ($A\in C$, $B\in(S\cup C)$), and $Z\in(S\cup C)$ is the start symbol; such that:

1) ($S\cap C=\varnothing$) (structural and collective nonterminals are non-overlapping);
2) $A\in S$ implies $(A\text{->}w)\in P$ for some w; $((A\text{->}w)\in P$ and $(A\text{->}w')\in P)$ implies $w=w'$ (there is exactly one production for each structural nonterminal);
3) $A>^+A$ holds for no $A\in C$ (no cyclic hierarchies);
4) $(A>B)\in H$, $(D>E)\in H$, $A\neq D$ implies $B\neq E$ (no multiple inheritance);

5) for each A∈(S∪C): Z=>*uAv=>+w, where u,v∈(S∪C∪T)* and w∈T* (no useless nonterminals).

Here we assumed syntactic derivation => defined as follows: vAu => vwu, if either (A->w)∈P or (w=B and (A>B)∈H), where A,B∈(S∪C), and v,u,w∈(S∪C∪T)*. We use the standard notation R+ for transitive closure and R* for reflexive transitive closure of relation R. The language generated by an OO-CFG (S,C,T,P,H,Z) is {w∈T* | Z =>+w}. In the following sections we assume that a language is generated by an OO-CFG.

Note that here we have required only those properties that make the object-oriented view sensible. In particular, we have not demanded that the grammar would be parsable in some way, or even unambiguous - these issues will be discussed in the sequel. For instance, even though cyclic hierarchies are forbidden, a structural nonterminal A may be cyclic: A=>+A is allowed.

As an example, consider the following fragment of an OO-CFG:

Statement > SimpleStatement | StructuredStatement
SimpleStatement > Assignment | ProcedureCall
StructuredStatement > IfStatement | WhileStatement | CompoundStatement
Assignment -> Identifier ":=" Expression
IfStatement -> "if" Expression "then" Statement "end"

The internal representation of sentence "if E then X:= 1 end" is depicted in Fig. 1.

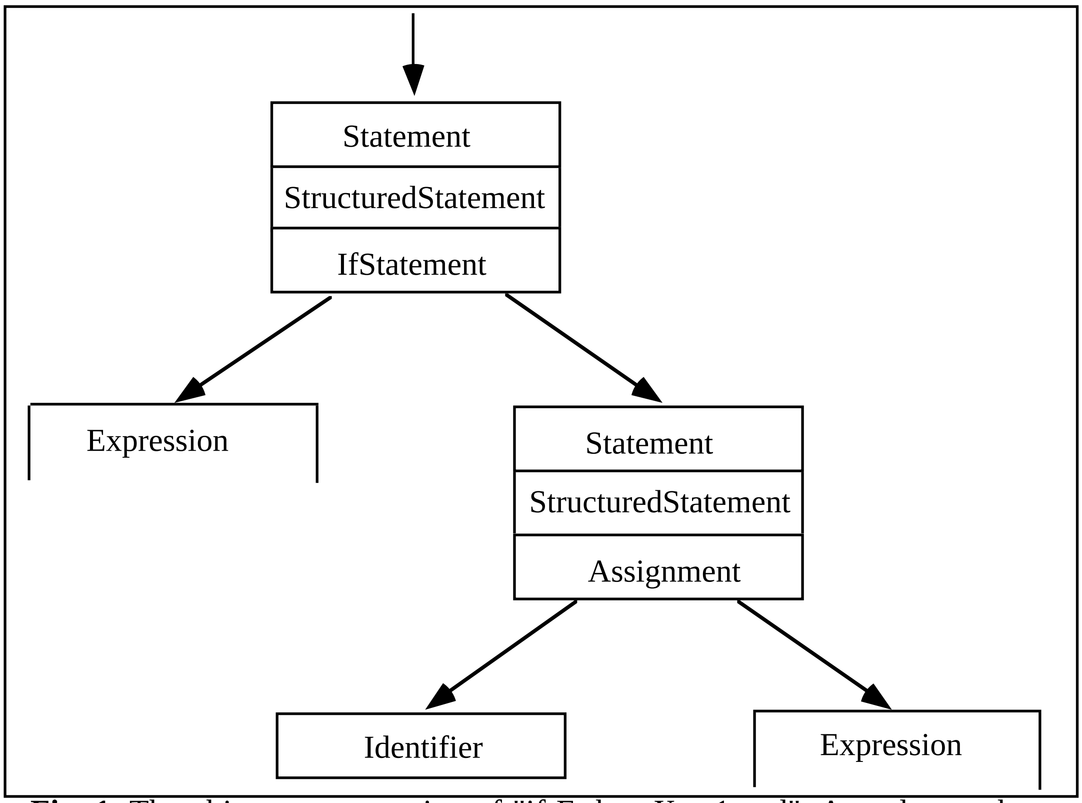

Fig. 1. The object representation of "if E then X:= 1 end". Arcs denote the values of syntactic attributes. Note that collective nonterminals give rise to upper layers of objects; the lowest layer belongs always to a structural nonterminal with syntactic attributes. The possible lower layers of Expression are not shown.

Lazy recursive descent parsing

Lazy recursive descent (LRD) was presented in [Kos90a] as a parsing method supporting the modularization of language implementations. The basic idea is to implement each nonterminal with a separate module in such a way that these modules are highly independent of each other:

all code that depends on the definition of a particular nonterminal is located in a single module. This is achieved by applying the traditional recursive descent method in a lazy manner: the required global information about the grammar - normally hard-coded in the parser - is computed at run-time, according to the needs of a given source text. In LRD this information consists of so-called "start trees" for nonterminal symbols; such a tree is a data structure showing the possible terminal starter symbols of the nonterminal, and the productions that have to be applied to generate a particular starter symbol. The leaves of the tree are the possible starter symbols, and the labels associated with the arcs on the path from a leaf node to the root give the productions (or actually their ordinal numbers with respect to the productions of the left-hand side nonterminal) leading to the appearance of the leaf symbol. If a nonterminal may produce the empty string, a particular "empty" leaf node is inserted into the tree, and the arc labels on the path from this leaf to the root indicate the productions that have to be applied when the empty string is produced. Hence, given nonterminal A and its starter symbol b, the productions on the path from b to the root of A’s start tree indicate the left edge of the syntax tree for string “b...” generated by A. We call these productions the *left-most* productions of A in that context.

LRD offers the same kind of intuitively understandable parsing code as the conventional recursive descent method, allowing semantic actions to be inserted easily. It has also been demonstrated [Kos90a] that LRD can be implemented as efficiently as conventional recursive descent, which is one the fastest known parsing methods. However, since the entire grammar is never known in the LRD method, certain new problems arise mainly in the screening of keywords; we will not discuss them here (for details, see [Kos90a]).

Consider the following grammar:

(1) A -> B C	(2) A -> "a"
(1) B -> D "d"	(2) B -> "b"
(1) C -> "c"	
(1) D -> "e"	(2) D ->

Here D produces the empty string in its second production. The production numbers are shown on the left. The start tree for A is given in Fig. 2. The labels of the branch nodes are not significant for the method, but they explain the structure of the tree. Note that D has a son which is a leaf with label "d" and arc number "2"; this means that A can start with a "d", and in that case the production number 2 must be applied for D (i.e., the empty string).

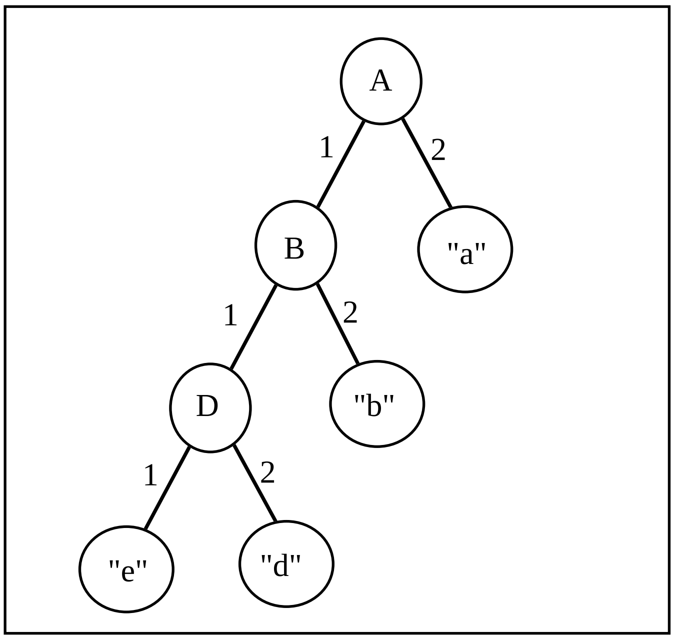

Fig. 2. The start tree for nonterminal A (see text).

Each nonterminal module provides two services: the actual parsing routine (PARSE) for the nonterminal, and a routine (PREPARE) computing the start tree for the nonterminal (stored in a variable of the nonterminal module). In both routines the services of the modules of the right-hand side nonterminals are used, but the form of these routines depend only on the productions of the nonterminal in question, and not on global grammar information. The parsing routine contains a case statement with a branch for each production of the nonterminal; a branch takes care of the parsing of the right-hand side of the corresponding production as usual. However, in contrast to conventional recursive descent the case labels are production numbers, not terminal symbol codes. The required production number is taken from a queue of production numbers maintained by the supporting system.

Functioning of lazy recursive descent parsing

An LRD parser runs in two modes called “left-down” and “right”. In the "right" mode a parser routine examines the start tree of the nonterminal and decides which left-most productions must be applied in that context. If the start tree is not yet constructed, the parser routine first calls the PREPARE routine to make the tree. This routine will in turn ask the nonterminals appearing on the right hand side to construct their start trees, if these trees are not yet made, and so forth. If a leaf node of the complete start tree matches with the current input, the production numbers along the path from the root to this leaf are stored in the queue. If there is no match but the start tree contains an "empty" leaf, this leaf is then selected, and the corresponding production numbers are stored in the queue; in other words, if the input does not match with the starters of the nonterminal that is supposed to come next, but this nonterminal is known to be able to produce the empty string, we assume that here the nonterminal indeed produces the empty string. If there is no match and no "empty" leaf, an error is reported. When the required production numbers have been stored in the queue, the parser routine switches mode (into "left-down") and enters the case statement, consuming the first production number from the queue, and proceeding with the code in the selected branch.

If the parser routine is entered in the "left-down" mode the start tree is not consulted, but instead the next production number is taken from the queue, and the corresponding case branch is selected directly. Only when the last production number in the queue has been consumed, the mode is again switched to "right". That is, in the left-down mode the parser proceeds along the left edge of the nonterminal's parse tree downwards as far as it can get, reaching finally the left corner of the parse tree. This results in stacking several activations of PARSE routines of nonterminals; the continuation of these activations affect the execution of the appropriate case branches. Each time a nonterminals's PARSE routine is called later in a case branch, the process is repeated.

Consider the example grammar given above, and suppose the input string is "dc". LRD parsing starts in "right" mode by calling A's PARSE (in module A). Since the start tree of A is not constructed, A's PREPARE is called to do this. The leaves of the resulting tree are compared with the current input (="d"). A match is found, and production numbers (1, 1, 2) are stored in the queue. The first number (1) is taken, causing the first production to be selected for A. This production starts with a call for B's PARSE, which is now entered in the "left-down" mode. The next number (1) is taken, selecting B's first production. There D's PARSE is called, still in "left-down" mode. The next number is taken (2), and the second production of D is selected. This is empty, and control returns to the code of B's right-hand side. There "d" is scanned, B's PARSE is completed, and control returns to A's right-hand side. There C's PARSE is called, now in the "right" mode, causing the start tree for C to be constructed. This tree consists of a single leaf "c" with arc "1"; hence 1 is stored in the queue and immediately consumed to select C's first (and only) production. Symbol "c" is scanned, and control returns to A's PARSE which is then completed.

This method obviously works for an LL(1) grammar: this condition guarantees that there is always a unique path from a given starter symbol to the root of a start tree. The fact that the LRD parser employs a default move leading to an empty string does not essentially change the behavior of the parser (except that certain non-LL(1) grammars will be parsable as well). In the following we assume that a language is generated by an OO-CFG satisfying the LL(1) condition. For simplicity, we ignore error recovery and assume that the analysis is aborted after the first error.

Notation for text synthesis

We adopt the notation for describing program synthesis introduced in [Kos90b]. Here we use only two “macro” structures, static if- and for-statements. These structures are static statements that are assumed to be executed by a preprocessor; as a result of the execution of such a statement, the statement will be replaced by some source text produced by the preprocessor. Any information the execution depends on must be static. The static statements are enclosed in brackets to separate them from the normal program text. Essentially, a program with such static statements is an algorithm for producing the actual program.

A static if-statement is of the form

```
[if E: ß]
```

where E is a (static) condition and ß is an arbitrary string. If the condition is true, this statement is replaced by ß in the final program; otherwise this statement has no effect on the program text. A variant with an else-part is given as

```
[if E: ß1 | else: ß2]
```

with obvious meaning: if E does not hold, the statement generates ß2. A static for-statement is of the form

```
[for X in S: ß]
```

where X is an identifier, S is a (static) ordered set, and ß is an arbitrary string. This statement is replaced by $ß_1$ $ß_2$ ... $ß_k$, where $ß_i$ is obtained from ß by replacing every occurrence of X with the i:th element of S (the cardinality of S is k).

3 Problem statement and solution outline

We aim at an incremental parser construction process in the object-oriented framework. More precisely, this process must satisfy the following requirements:

1) *Nonterminal-class mapping*

Each nonterminal gives rise to one or more *implementation classes*. One of these classes specifies the instances of the nonterminal in the source text; we call this the *actual nonterminal class*. All semantic processing is specified as methods of the actual nonterminal classes. In addition to the actual class, other classes may be given for a nonterminal for technical reasons. Only these implementation classes are dependent of the grammar.

2) *Class hierarchy*

The class hierarchy of actual nonterminal classes follows the object-oriented interpretation of the context-free grammar of the language, discussed in Section 2.

3) *Internal representation of the source*

An internal representation of the source text is constructed as objects which are instances of actual classes of structural nonterminals.

4) *Incrementality*

> Changing the syntactic rule of a structural nonterminal and adding or deleting a child nonterminal for a collective nonterminal have minimal effect on the implementation classes of other nonterminals. Note that since other than implementation classes are independent of the grammar, they cannot be affected by changes in the grammar.

Let us first consider a straightforward recursive descent solution. Assume a structural nonterminal N, with rule N->X_1...X_k. Since the right-hand side of the rule can be viewed as a creation instruction for N, it is natural to put the corresponding parsing code into the creation operation (below: `analyze`) of the actual class of N. When N appears on the right-hand side of another structural nonterminal, it is sufficient to instantiate this class and store the resulting object reference into a syntactic attribute. In Eiffel, the actual class of N would therefore look like:

```
class N
  inherit
    SOURCE;       -- provides a global source handler
    [if N has a collective parent nonterminal M: M
      |else: NOTION]
  creation analyze
  feature
    ...
    [for i in 1..k:
      [if Xi is nonterminal symbol:      -- syntactic attributes
        c_i: Xi; ]]
    analyze is do
    [for i in 1..k:
      [if Xi is terminal symbol:
        scanner.scan(Xi); ]
      [if Xi is nonterminal symbol:
        !!c_i.analyze]]                  -- instantiation of class Xi
    end
  end
```

Here `NOTION` is assumed to be the superclass of all nonterminal classes (i.e. the root of the nonterminal class hierarchy); we will not define it here in detail. `SOURCE` is a class providing access to the source text through a scanner; we leave this class unspecified, too. In Eiffel, the instantiation operation is given in the form !S!x.p; this results in creating an object of class S, storing its reference in variable x, and calling the (creation) operation p on this object. If S is missing (as above), the class of the object will be the (static) class of x.

If there were only structural nonterminals, this would be a trivial solution to our problem: the classes need to know only the names of their component structures. Problems arise in the case of a collective nonterminal, say N>X_1|...|X_p. If N appears on the right-hand side of a production, it is not the actual class of N that has to be instantiated, but some of its descendant classes corresponding to a structural nonterminal. Assuming the grammar is LL(1), the descendant class to be instantiated is uniquely determined by the current input symbol. In Eiffel this could be implemented by constructing for every collective nonterminal a special class providing the selection of the correct descendant class on the basis of the input. For N, this class could look like:

```
class N_SELECTION
  feature
    some_N: N is
    do
      -- let the structural descendant nonterminals
      -- of N be S1, S2, ... , Sp:
      if
      [for i in 1..p-1:
        input belongs to First(Si) then !Si!Result.analyze elsif]
        input belongs to First(Sp) then !Sp!Result.analyze
```

```
         [if Sj =>+ empty for some j, 1≤j≤p:
            else !Sj!Result.analyze end
         |else: end]
      end
   end
```

where First(S) denotes the set of starter terminal symbols for S. Result is a pseudo variable representing the value of a function. In addition to this class, there is the actual class of the collective nonterminal N consisting of user-defined semantic features of N; we do not specify it here. The code for a structural nonterminal N->X_1...X_k must now be revised:

```
class N
   inherit
      SOURCE;
      [if N has a collective parent nonterminal M: M
         |else: NOTION]
      [for i in 1..k:
         [if Xi is collective nonterminal:
            ; Xi_SELECTION]]
   creation analyze
   feature
      ...
      [for i in 1..k:
         [if Xi is nonterminal symbol:      -- syntactic attributes
            c_i: Xi; ]
      analyze is do
      [for i in 1..k:
         [if Xi is terminal symbol:
            scanner.scan(Xi); ]
         [if Xi is structural nonterminal:
            !!c_i.analyze; ]                 -- instantiation of Xi
         [if Xi is collective nonterminal:
            c_i:= some_Xi; ]]
      end
   end
```

This simple solution does not satisfy the incrementality requirement because class `N_SELECTION` has to be revised every time the class hierarchy below N changes, or the syntactic form of some of the structural nonterminals Si changes; similarly, for a structural nonterminal N, class `N` has to be modified if some of the components of N changes its kind from structural to collective or vice versa.

Our general strategy to reduce the dependencies among classes is to increase the dynamic character of the solution, without losing the basic intuitively appealing form of the top-down deterministic parser. Note that the text of the classes make use of three kinds of static information we want to get rid of: the starter symbols of nonterminals (and the fact that a nonterminal may produce the empty string), the class hierarchy, and the kind of nonterminals (structural or collective).

We employ two methods to eliminate the static presence of this information. First, to present the starter symbols dynamically we apply the LRD technique. Each collective nonterminal is associated with a list of starter symbols (called *starter list*) computed at run time; in this list every starter symbol is accompanied by some (dynamic) information giving the class to be instantiated. Second, to present as much class information as possible dynamically we use metaobjects. In general, a metaobject is an object whose main purpose is to serve as a creator of other objects. A metaobject is therefore a natural dynamic counterpart of a class: class instantiation can be viewed as sending a message to the corresponding metaobject. In our case each nonterminal will be represented at run time by a metaobject whose only task is to create instances of the actual nonterminal class. To summarize, for each nonterminal we need at least two implementation classes: the class of the metaobject (called *metaclass*) and the actual non-terminal class.

Since a starter list provides the essential information for creating objects, it will be stored in an attribute of the metaobject representing the nonterminal in question. The nature of the information associated with each starter symbol, giving the class to be instantiated, becomes now obvious: it is a reference to the metaobject of the (structural) nonterminal that has to be instantiated in that context.

The use of metaobjects unify the treating of different nonterminals appearing on the right-hand side of a production: independently of the kind of the nonterminal (structural or collective), the metaobject of the nonterminal is asked to create the appropriate instance of an actual nonterminal class. In the case of a structural nonterminal, the metaobject simply calls the creation operation of the actual nonterminal class and returns a reference to the object thus created; in the case of a collective nonterminal the metaobject consults its starter list, finds a reference to the appropriate metaobject, and delegates the creation task to this object.

To illustrate the use of the metaobjects, consider the following OO-CFG:

```
A > A1 | A2             B > B1 | B2            D1 -> "e"
A1 -> B C               B1 -> D "d"            D2 ->
A2 > A21 | A22          B2 -> "b"
A21 -> "a"              C -> "c"
A22 -> "x"              D -> D1 | D2
```

The metaobjects and their starter lists are shown in Table 1.

	starter list						
metaobject	a	b	c	d	e	x	ε
A	A21	A1		A1		A22	
A1		A1		A1			
A2	A21					A22	
A21	A21						
A22						A22	
B		B2		B1			
B1				B1	B1		
B2		B2					
C			C				
D					D1		D2
D1					D1		
D2							D2

Table 1. Metaobjects and their starter lists for the example grammar.

Table 1 should be read as follows: first line indicates that metaobject of A has a starter list with symbol "a" associated with the metaobject of A21, symbol "b" associated with the metaobject of A1, etc. Note that the symbols in the starter list of the metaobject for a structural nonterminal are always associated with the metaobject itself. The row for a collective nonterminal is constructed as a union of the rows of its child nonterminals. ε denotes the empty string: if a nonterminal has an entry in this column, it may produce the empty string.

Suppose that the input is "dc". Parsing starts by asking the metaobject of A to create an appropriate object. This metaobject consults its starter lists, finds an entry for the next input symbol ("d"), and delegates the creation task to the metaobject of A1. This metaobject will in turn directly instantiate A1's actual class, activating thus the parsing code for the right-hand side BC. Then B's metaobject will be the next target for creation request; this metaobject again finds "d" in its starter list and delegates the job to B1's metaobject. This leads to the instantiation of B1, which in turn activates the code for D "d". D's metaobject cannot find the next input symbol in its starter list, but since it has an entry for ε, it assumes that D must be empty in this context, and delegates the job to the metaobject of D2. Class D2 is instantiated, and control returns the instantiation code of B1 where "d" is scanned. Execution continues in A1's instantiation code, requesting C's metaobject to instantiate C. Finally "c" will be scanned, and parsing is completed.

The OO-CFG form makes the application of the LRD technique remarkably simple. Because alternation is described via subclassing, the start tree can be linearized into a list: the tree structure is implicitly determined by the class hierarchy. There is no need to remember the productions that have to applied in the case of a particular starter symbol; here a starter symbol yields merely the lowest class level (corresponding to a structural nonterminal), and the selections leading to this alternative are automatically reflected in the superclass layers of the created object.

Assuming that the starter lists have been constructed, this solution leads to the desired effect: the code for the implementation classes depends only on the definition of the nonterminal in question, and not on the other nonterminals. In particular, since the code for the creation operation provided by the metaobject is actually constant, adding or removing a child for a collective nonterminal has no effect on the code of the metaclass. However, the problem is partially pushed aside: the construction of the starter lists may give rise to dependencies as well.

Note that the starter symbols of a collective nonterminal consists of the starter symbols of all the descendant nonterminals. Hence the construction of the starter lists can be done in two ways, bottom-up or top-down: either a metaobject of a child nonterminal notifies the metaobject of its parent nonterminal about its existence and starter symbols, or a metaobject of a parent nonterminal consults the metaobjects of its child nonterminals when constructing its own starter list. However, both methods have drawbacks. The bottom-up direction is not suitable for a fully lazy parser in which metaobjects are created only when needed. This is due to the fact that in recursive descent parsing nonterminals will be used in the top-down order; but a child nonterminal cannot be used before it has informed its parent - which it will not do before it is used! Therefore the bottom-up direction implies that all the metaobjects are created before the actual parsing begins, independently of the input string. On the other hand, the top-down direction is in conflict with incrementality because it assumes that the metaobject of a collective nonterminal knows the identity of the metaobjects of its children; essentially, this implies that the metaclass contains static information about the subclasses of the actual nonterminal class.

The bottom-up direction guarantees incrementality, which is our primary aim, but it imposes a particular requirement on the implementation language. To be fully incremental, a child nonterminal must notify the parent about its existence using some code in its own implementation classes. However, since these classes are unknown to the other classes and cannot therefore be instantiated, such code cannot be executed unless there is some way that allows a class to execute code prior to its instantiation. This is an example of the so-called problem of unexpected subclasses; it has been demonstrated [KoV92] that in Eiffel this problem is in principle unsolvable whereas C++ supports this facility through static objects and their constructor functions. Hence in the following we will use the top-down approach in the Eiffel solution, offering a fully lazy but only partially incremental technique: the implementation classes of a collective nonterminal must know the names of the child nonterminals, but not the entire class hierarchy below that nonterminal (as in the simple solution above). Since the classes cannot be made fully independent in Eiffel, this solution can be considered as optimal as possible with respect to incrementality. In contrast, the given C++ solution employs the bottom-up approach, and yields a fully incremental technique. This solution is partially lazy: all the metaobjects exist statically independently of the input string, but their starter lists are computed only when needed.

4 Eiffel solution: partial incrementality

We start by defining the general, grammar-independent classes used by all nonterminals. The information concerning the possible starter symbols of a nonterminal are stored in a list whose element is given by class `STARTER_SYMBOL`:

```
class STARTER_SYMBOL
  feature
    terminal: STRING;
    token_code: INTEGER;
    metaobj: META_NONTERMINAL;
  end
```

Here `metaobj` refers to the metaobject of a structural nonterminal; this nonterminal has to be instantiated when the corresponding starter `terminal` is found in the input string while beginning the parsing of the nonterminal whose metaobject has this item in its list of starters. Attribute `token_code` gives the (integer) code for the terminal category (keyword, identifier, integer number, real number, string); `terminal` is significant only if the code denotes a keyword (in which case `terminal` gives the actual keyword string).

```
class STARTER_LIST
  inherit LIST[STARTER_SYMBOL]; SOURCE
  feature
    eps: BOOLEAN;
    eps_meta: META_NONTERMINAL;
    set_eps(m: META_NONTERMINAL) is
    do
      eps:= True; eps_meta:= m
    end;
    add_keyw(kw: STRING; meta: META_NONTERMINAL) is
    do
    ... add new item with terminal=kw, token_code=0, metaobj=meta
    end;
    add_token(tc: INTEGER; meta: META_NONTERMINAL) is
    do
    ... add new item with token_code=tc, metaobject=meta
    end;
    add_list(s: STARTER_LIST; meta: META_NONTERMINAL) is
    do
    ... merge s with this list; if there are duplicates,
    ... report a grammar error;
    ... for every new item, set metaobj=meta (independently
    ... of the original value of metaobject in s)
    end;
    join(s: STARTER_LIST) is
    do
    ... merge s with this list; if there are duplicates,
    ... report a grammar error;
    ... for every new item, retain the original value of
    ... metaobj-attribute
    end;
    match: META_NONTERMINAL is
    do
    ... if one of the list items matches with the current
    ... input, return the metaobject representing the
    ... structural nonterminal that ought to be
    ... instantiated (i.e. the value of metaobj-attribute
    ... of that item); if none of the items matches, report an
    ... error in the input text
    end
  end

deferred class META_NONTERMINAL
  feature
    starters: STARTER_LIST;
    make: NOTION is deferred;  - virtual function
  end
```

The operation join is used to define the starter symbols of a collective nonterminal as determined by its descendants. The operation add_list adds new starter symbols to a structural nonterminal.

Class `META_NONTERMINAL` is the common abstract super class for all metanonterminal classes. For each nonterminal N, the following standard implementation class is given:

```
class N_ACCESS
  feature
    metaobj_N: META_N is once
      Result.Create
    end
  end
```

The body of a once function is executed only once, when the function is called for the first time; subsequent calls return the value computed by the first activation. Once functions are the standard way to access global objects in Eiffel; in this case the global object is the metaobject representing the nonterminal.

For each nonterminal N, two additional implementation classes `N` and `META_N` are given. The former is the actual nonterminal class, while the latter describes the metaobject for N (i.e. the object denoted by `metaobj_N` of class `N_ACCESS`).

In the case of a collective nonterminal N, classes `N` and `META_N` are defined below. Let the subclasses of `N` be `N1`, ... , `Np` ($p \geq 0$) (that is, $N > N_1 | ... | N_p$).

```
class META_N
  inherit META_NONTERMINAL;
    [for j in 1 .. p:
    Nj_ACCESS;]
  creation init
  feature
    make: N is                    -- definition of virtual function
    do
      Result?= starters.match.make; -- never fails!
    end;
    init is do
      !!starters;
      [for j in 1 .. p:
        starters.join(metaobj_Nj.starters);]
    end
  end

class N
  inherit
    [if N has a collective parent nonterminal M: M
      | else: NOTION]
  feature
  --  ... possible user-defined semantic attributes and methods
  --  ... no creation operation required
  end
```

Here we make use of the “inverse assignment” of Eiffel: in assignment “a?=b“, a’s class must be a descendant of b’s static class; an implicit run time check guarantees that the dynamic class of b. Note that if N has a collective parent nonterminal in the grammar, `NOTION` need not be inherited because it will be inherited indirectly through an ancestor class. Here the creation operation of `META_N` constructs the starters list, using (and possibly creating; recall the once function) the metaobjects of the subclasses. Note that this class needs to know the names of its subclasses.

In the case of a structural nonterminal, the classes have the following structure. Let the syntactic rule for N be $N \to X_1\ X_2\ ...\ X_p$ ($p \geq 0$), where X_i is the first terminal symbol in the sequence $X_1\ X_2\ ...\ X_p$. If there are no terminal symbols, let i=p+1.

```
class META_N
  inherit META_NONTERMINAL
      [for j in 1 .. p:
      ; Xj_ACCESS]
  creation init
  feature
    make: N is
    do
      !!Result.analyze
    end
    init is
    do
      !!starters;        -- create starter list
      [for j in 1 .. i-1:
        starters.add_list(metaobj_Xj.starters, Current);
        if metaobj_Xj.starters.eps then]
          [if i=p+1:
            starters.set_eps(Current);]
          [if Xi is keyword:
            starters.add_keyw(Xi, Current);]
          [if Xi is token class:
            starters.add_tc(category code for Xi, Current);]
      [for j in 1 .. i-1:
        end]
    end
  end

class N
  inherit
    [if N has a collective parent nonterminal M: M
      | else: NOTION];
    SOURCE
    [for j in 1 .. p:
      ; Xj_ACCESS]
  creation analyze
  feature
    [for j in 1 .. p:
      [if Xj is nonterminal symbol:
        s_j: Xj;    -- structural attribute for component Xj ]]
    ... other possible user-defined attributes & methods ...
    analyze is
    do
      [for j in 1 .. p:
        [if Xj is keyword:
          scan(Xj);]
        [if Xj is token class:
          scan_tc(category code for Xj);]
        [if Xj is nonterminal symbol:
          c_j:= metaobj_Xj.make;]]
    end
  end
```

In this case, the creation operation of `META_N` is more complicated: the starter symbols of N are computed by processing the right-hand side of N's rule from left to right, until a terminal symbol is encountered. At run-time, the processing is completed when the first item not producing the empty string is encountered. If no such item is found, N is marked to produce the empty string itself. For a structural nonterminal N, the only task of the `make` operation of `META_N` is to create an instance of `N`, activating the creation operation of `N` that eventually executes the parsing. Possible parse-time semantic actions (e.g. symbol-table management, type checking) will be inserted into the code of the latter operation.

The creation operation of `N` (`analyze`) contains the traditional recursive descent parsing code for nonterminal N, except that the internal representation of the source is constructed as well:

references to the objects representing component structures are stored in the syntactic attributes s_j. Note that in this code all nonterminal symbols appear in the same way, independently of their kind (collective or structural) or syntactic rules; hence any changes in the definition of the other nonterminals have no effect on this code. All desired semantic actions can be inserted into this code; note that for instance multiple passes over the source are easily implemented, due to the existence of the objects representing the source in an abstract form.

A subset of Pascal with 75 nonterminals was implemented in Eiffel 2.3 using this technique [Fra92]. We employed no automated generation tools, but nevertheless the construction of the parser was remarkably simple and straightforward: using general class templates the language dependent classes could be produced easily. Unfortunately, due to the pecularities of Eiffel 2.3, especially its string handling, the performance of the resulting parser was rather poor (in fact, the time consumption was found to be nonlinear with respect to the length of the input - a result which can only be explained by the internal properties of the underlying system).

5 C++ solution: full incrementality

Designed as a hybrid language, C++ allows more flexible programming than “pure” statically typed object-oriented languages. In particular, we make use of static objects and their constructor functions: static objects are created and initialized prior to the execution of the main program. Recall that in C++ functions must be explicitly declared **virtual**; a virtual function is *pure* if it is given a null body, corresponding to the Eiffel term **deferred**.

The main difference between Eiffel and C++ solutions is the handling of descendant nonterminals of a collective nonterminal. The algorithms are otherwise identical, except for the way nonterminals are defined as descendants of some collective parent. In the Eiffel solution, the set of direct ascendants is known statically, and hard-coded as part of the class definition of the parent. In C++ solution, the descendants are determined dynamically: a new descendant notifies itself its parent at run time. Note that in Eiffel this would not work because there is no way in which one could add any separately compiled code to a complete system such that this code would be sometimes executed by the system - unless we change the existing classes of the system. In C++ this is possible e.g. using constructor functions (i.e. initialization functions) of static objects. We will not repeat here those parts that are essentially identical to the Eiffel solution; these parts include the language independent classes STARTER_SYMBOL and STARTER_LIST.

The class NOTION represents the actual language constructs, and after the metaobject has selected the appropriate syntactic alternative, the constructs parse themselves. The abstract class NOTION postpones the implementation of the parser function until a concrete structural nonterminal is introduced.

```
class NOTION {  // abstract class
public:
  virtual void analyze () = 0;    // the actual parsing of the
                                  // language construct
}; // NOTION
```

The class META_NONTERMINAL needs the following functions. The function make() is defined as pure, and must be redefined both in collective and structural nonterminal classes. The constructor function has an initialization parameter that defines the parent nonterminal. The parent is non-null for any descendant of a collective nonterminal; a nonterminal may also first appear as a component of a structural nonterminal. The new descendant is added to a dynamic list of descendants. The actual insertion is handled by the function new_descendant(); the details of its implementation are omitted here.

```
class META_NONTERMINAL {   // abstract class
public:
  virtual NOTION* make () = 0; // make an instance of some descendant
  virtual void init (); // calculate starter symbols for descendants
```

```
        META_NONTERMINAL (META_NONTERMINAL* parent)
        {  // define parent if any
          if (parent) parent->new_descendant (this);
        }
      private:
        STARTER_LIST *starters;          // associative list of descendants
        void new_descendant (META_NONTERMINAL* descendant)
        {
          // ... add new dynamic descendant nonterminal ...
        }
      }; // META_NONTERMINAL
```

The function `init()` is virtual, too, but it is given a default implementation that applies to collective nonterminals. The default behaviour merges the starter symbols of descendant nonterminals with the starter list of the parent.

```
      void META_NONTERMINAL::init () {
        // calculate starter symbols of collective nonterminal
      } //  init
```

For structural nonterminals, this function must be redefined. The default implementation of `init()` is logically similar to the Eiffel solution and is omitted here.

The abstract classes `NOTION` and `META_NONTERMINAL` are predefined as part of the core system. Next we consider the classes representing the constructs of a language. They can be either collective or structural.

For a collective nonterminal N, we define two classes, the class `N` to represent the semantic language construct and its properties, and the class `META_N` to represent the syntactic alternatives.

```
      class N: public M {  // abstract class;  if no parent: M = NOTION
      public:
        possible user-defined semantic attributes
        // analyzed ()  remains undefined, so N cannot be instantiated
      }; //  N
```

The class `N` defines only a semantic layer corresponding the collective language notion N. This layer could well be empty, and thus be omitted. The metaobject is defined as follows.

```
      class META_N: public META_NONTERMINAL {  // concrete class
      public:
        NOTION*  make () { return starters.match.make (); }
        META_N (META_NONTERMINAL* parent) : (parent) {}
      }; // META_NONTERMINAL
```

The constructor funtion of `META_N` simply passes the identity of the parent metaobject (if any) to the superclass constructor. A new syntactic alternative is thus added to a collective nonterminal. The function `init()` is already defined in the super class `META_NONTERMINAL`.

Lastly, we create the metaobject. The metaobject needs to know the identity of its parent in order to introduce itself to it. The parent is accessed as a globally defined entity.

```
      META_N  meta_N (meta_M);  // if no parent, meta_M = 0
```

Next we consider the definition of structural nonterminals. In this case, the class `N` is a concrete class.

```
      class N: public M {  // if no parent M = NOTION
      public:
        N () { analyze (); }
        void analyze () { ... actual parsing of language construct ...}
```

```
   ...  possible user-defined semantic attributes
};
```

The definition of `analyze ()` is identical to the one in Eiffel. Now the instantiation of `N` becomes possible. The instances of `N` are created by the corresponding metaobject:

```
class META_N: public META_NONTERMINAL {  // concrete class
public:
   NOTION*  make () { return new N; }
   META_N (META_NONTERMINAL* parent) : (parent) {}
   void init ();   //  similar to Eiffel solution ...
}; // META_N
```

The function `make()` delegates the parsing to the object itself. The virtual function `init()` must be redefined to handle starter symbols of a structural nonterminal. The algorithm is identical to the one given for the Eiffel solution.

A structural metaobject is created in the same manner as a collective metaobject.

```
META_N  meta_N (meta_M);  // if no parent, meta_M = 0
```

C++ does not impose any order on the initialization of static objects from different translation units; this complicates somewhat the initialization of static metanonterminals. This problem has been thoroughly discussed in [KoV92]. The calculation of the starter symbols may occur either in a pure incremental manner, as in Eiffel solution, or during special initialization phase activated by the main function. Since the metaobjects are created as static objects prior to the execution of the main function, all the required information to determine the starter symbols is already available.

6 Discussion

Although our primary motivation has been the requirements of automated parser synthesis, the incremental parser construction techniques discussed here have clear advantages independently of the production method. These techniques support stepwise language implementation: the implementation can start with a small core language, and gradually add new features represented by new nonterminal subclasses. At least from the parsing point of view, the introduction of new subclasses can be done without touching the existing classes at all (C++), or with only some minor changes in the superclasses (Eiffel). In experimental implementation of a new language, individual language features can be easily changed syntactically - only the relevant class has to be revised. Subset languages can be formed simply by removing some of the subclasses representing certain syntactic alternatives. An internal representation of the source program is constructed in a natural way, and semantic processing can be added in an intuitively satisfactory way as methods of the nonterminal classes. Finally, the implementation software is easy to understand and maintain, because its structure reflects the application domain, a language generated by a context-free grammar; this is implied directly by the object-oriented programming paradigm. We claim that the use of metaobject-driven parsing is as natural for object-oriented programming as procedure-based recursive descent is for traditional programming.

Acknowledgements

We wish to thank Mr. Alexander Franz from the Technical University of Dresden for carrying out the experimental implementation of the Eiffel solution. This work is supported by the Academy of Finland through grant 1061120.

References

[Bor91] Borland C++ Version 3.0. Borland International, USA, 1991.

[CNS87] Christ-Neumann M.-L., Schmidt H.-W.: ASDL - An Object-Oriented Specification Language for Syntax-Directed Environments. In: Proc. of ESEC '87, LNCS 289, Springer-Verlag 1987, 71-79.

[Fer89] Ferber J.: Computational Reflection in Class based Object-Oriented Languages. In: Proc. of OOPSLA ‘89, Sigplan Notices 24,10 (Oct. 1989), 317-326.

[Fra92] Franz A.: The Implementation of an Incremental Parser in Eiffel. Internal report, Department of Computer Science, University of Tampere, September 1992.

[Gro84] Grossmann R., Hutschenreiter J., Lampe J., Lötzsch J., Mager K.: Depot2a - Metasystem für die Analyse und Verarbeitung Verbundener Fachsprachen. Anwenderhandbuch, Sektion Mathematik, Technische Universität Dresden, 1984.

[Gyi88] Gyimóthy T., Horváth T., Kocsis F., Toczki J.: Incremental Algorithms in PROF-LP. In: Proc. of Workshop on Compiler-Compilers, Lecture Notes in Computer Science 371, Springer-Verlag 1989, 93-102.

[Hed89] Hedin G.: An Object-Oriented Notation for Attribute Grammars. In: Proc. of the European Conference on Object-Oriented Programming (ECOOP '89), Nottingham, 1989. The British Computer Society Workshop Series, Cambridge University Press 1989, 329-345.

[HeR75] Heindel L.E., Roberto J.T.: Lang-Pak - An Interactive Language Design System. Elsevier 1975.

[HKR89] Heering J., Klint P., Rekers J.G.: Incremental Generation of Parsers. In: Proc. of ACM Sigplan '89 Conference on Programming Language Design and Implementation, Portland, Oregon, 1989. ACM Sigplan Notices 24,7 (1989), 179-191.

[Hor90] Horspool R.N.: Incremental Generation of LR Parsers. Journal of Computer Languages, 15, 4 (1990), 205-223.

[JKP91] Järnvall E,. Koskimies K., Paakki J.: The Design of Tampere Language Editor (TaLE). Report A-1991-10, Department of Computer Science, University of Tampere, Finland, 1991.

[Kli91] Klint P.: A Meta-Environment for Generating Programming Environments. In: Proc. of METEOR Workshop on Methods Based on Formal Specification. Lecture Notes in Computer Science 490, Springer-Verlag, 1991, 105-124.

[Kos88] Koskimies K.: Software Engineering Aspects in Language Implementation. In: Proc. of Workshop on Compiler Compilers, Lecture Notes in Computer Science 371, Springer-Verlag 1988, 39-51.

[Kos90a] Koskimies K.: Lazy Recursive Descent Parsing for Modular Language Implementation. Software Practice & Experience 20,8 (August 1990), 749-772.

[Kos90b] Koskimies K.: Techniques for Modular Language Implementation. Acta Cybernetica 9,3 (1990), 193-209.

[Kos91] Koskimies K.: Object-Orientation in Attribute Grammars. In: Proc. of SAGA Summer School on Attribute Grammars and their Applications, June 1991. Lecture Notes in Computer Science 545, Springer-Verlag, 1991, 297-329.

[KoV92] Koskimies K., Vihavainen J.: The Problem of Unexpected Subclasses. To appear in Journal of Object-Oriented Programming, 1992

[LMN88] Lehrmann Madsen O., Nørgaard C.: An Object-Oriented Metaprogramming System. In: Proc. 21st Annual Hawaii International Conference on System Sciences (B.D. Shriver ed.), 1988, 406-415.

[Lie87] Lieberman H.: Reversible Object-Oriented Interpreters. In: Proc. of the European Conference on Object-Oriented Programming (ECOOP '87), Lecture Notes in Computer Science 276, Springer-Verlag 1987, 11-19.

[Mag90] Magnusson B., Bengtsson M., Dahlin L.-O., Fries G., Gustavsson A., Hedin G., Minör S., Oscarsson D., Taube M.: An Overview of the Mjølner/ORM Environment: Incremental Language and Software Development. Report LU-CS-TR:90:57, Department of Computer Science, Lund University, 1990. Also in Proc. TOOLS '90, Paris 1990.

[Mey88] Meyer B.: Object-Oriented Software Construction. Prentice-Hall 1988.

[Mey92] Meyer B.: Eiffel - The Language. Prentice-Hall 1992.

[Ten88] Tenma T., Tsubotani H., Tanaka M., Ichikawa T.: A System for Generating Language-Oriented Editors. IEEE Trans. on Software Engineering 14,8 (August 1988), 1098-1109.

[WuW92] Wu P-C., Wang F-J.: An Object-Oriented Specification for Compiler. Sigplan Notices 27,1 (Jan 1992), 85-94.