# LANGUAGE IMPLEMENTATION MODEL IN TaLE

Esa Järnvall and Kai Koskimies

**DEPARTMENT OF COMPUTER SCIENCE**
**UNIVERSITY OF TAMPERE**

**REPORT A-1993-1**

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1993-1, FEBRUARY1993

# LANGUAGE IMPLEMENTATION MODEL IN TaLE

Esa Järnvall and Kai Koskimies

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

ISBN 951-44-3324-6
ISSN 0783-6910

# Language Implementation Model in TaLE

Esa Järnvall and Kai Koskimies
Department of Computer Science, University of Tampere
Box 607, 33101 Tampere, Finland
email: koskimie@cs.uta.fi

**Abstract**

TaLE (Tampere Language Editor) is a specialized program editor for developing implementations of textual languages. The system assumes an object-oriented programming language (currently Eiffel), and supports the development of language implementations following a particular model. The basic object-oriented language implementation model, including integrated methods for lexical analysis, syntactic analysis, name analysis, error recovery, and the construction of an internal (object) representation is discussed. A central requirement implied by the incremental character of the system is that the information concerning the language is distributed among the classes representing structural units of the language (like nonterminals), so that changes in the definition of a particular unit have minimal effects on the classes representing other units. A general method for solving LL(1) parsing conflicts using semantic information is included in the model.

## 1 Introduction

TaLE (Tampere Language Editor) is a new system supporting the editing of classes that take part in the implementation of a textual language. The intended typical application areas of TaLE are the implementations and processing of various special-purpose languages; hence TaLE emphasizes user-interface and software engineering aspects, rather than the properties of a formal metalanguage - in fact, there is no metalanguage in the traditional sense. We consider TaLE as a program editor comparable to, say, resource editors used for creating graphical user interfaces. An overview of TaLE can be found in [JäK93].

TaLE supports the user by offering a graphical, specialized view of the essential classes taking part in the implementation. The use of an object-oriented language makes it possible to establish a one-to-one correspondence between the intuitive notions of the source language and the software units; this is a key requirement for a language implementation system based on the editing paradigm. The difference between TaLE and a general object-oriented programming environment (say, Smalltalk) is that TaLE takes automatically care of the uninteresting parts of the software concerning standard implementation techniques, and hides them from the user. The classes are accessible for the user indirectly through various high-level, mostly graphical views. The hiding of the uninteresting parts implies that the non-visible parts associated with a particular language structure are allowed to depend only on the structure itself; this is essential because otherwise the system should regenerate (and recompile) the non-visible parts of other classes after the user has edited some class.

In this paper we present the implementation model implied by these requirements in TaLE. It should be emphasized that this model is not the same as the *language* model of TaLE, as seen by the user of this system (discussed e.g. in [JäK93]); rather, we discuss here how the language model can be implemented in Eiffel. Most of the aspects considered here are not even visible for the user of TaLE, and probably not interesting for her. A characteristic feature of the implementation model is that all the information concerning the source language is distributed among classes representing particular language structures: for instance, the classes corresponding to a nonterminal contain statically exactly the information associated with that particular nonterminal, no more, no less. Since there are no other language-dependent classes in addition to these, the incrementality requirement is satisfied: changing the definition of one nonterminal necessitates the regeneration of the classes specific to that nonterminal, but no other classes[1].

The TaLE implementation model is based on the metaobject-directed LL(1) parsing technique presented in [KoV92b]. This technique is generalized and improved in TaLE in several ways:

1) keywords, token categories and nonterminals are treated in a uniform manner, allowing e.g. user-defined token categories;

2) LL(1) parsing is generalized allowing nonterminal symbols as starter symbols used for resolving look-ahead conflicts;

3) iteration, set, and optional structures are allowed in the syntax;

4) a look-ahead problem implied by the original technique is solved in a general way;

5) a general technique for using name analysis to support parsing is included in the model;

6) a simple but practical error recovery technique is developed.

We concentrate here on these improvements (for a more detailed discussion of the basic ideas of metaobject-directed parsing, see [KoV92b]).

*Related work*

Besides [KoV92b], object-oriented language processing has been discussed in several papers recently (e.g. [CNS87], [Gro90], [Kos91], [Hed92], [Gra92], [AlR92], just to mention few). Our implementation model is different from these in that we are interested in incremental language implementation techniques in a framework provided by a general, statically typed object-oriented programming language - this is a consequence of the TaLE approach in which the user manipulates the actual implementation classes without a mediating metalanguage. Hence, although the TaLE editor automatically inserts some "standard" parts of the edited classes, our techniques are sensible in hand-written language implementations as well as in automated language implementation: the inserted parts are understandable enough to be written by a human. We do not use any global, interpreted files (like scanning and parsing tables), but all information and capabilities concerning the language are distributed among the classes representing individual language structures (nonterminals, tokens).

A particular requirement in our model is incrementality: each class knows (statically) only the information directly associated with the language structure in question. This makes it possible to develop and test a language piecewise one structure at a time. Incremental language implementation techniques have been investigated (in non-object-oriented environments) by several other authors, too. In particular, incremental LR or LL parser construction methods have been discussed in [HKR89], [Hor90], and [Gyi88]. All these works consider incrementality in the context of a metalanguage-based compiler writing system with table-driven parsers, whereas we use directly an object-oriented progamming language and techniques in which the parsing code is distributed among the nonterminal-specific classes. A program-formed incremental LL(1) parser construction technique is presented in [Kos90]; this technique has been a starting point in [KoV92b] and is therefore a basis of our work, too.

It should be emphasized that we are interested in "meta-time" incrementality rather than in "compile-time" incrementality; i.e. we aim at the incremental construction of the language processor, not at the incremental construction of the programs processed through that processor. With some exceptions (e.g. Orm [Mag90]), these issues are usually studied separately.

---

[1]We consider here only the standard syntactic and semantic analysis. If the user introduces additional semantic attributes for a nonterminal, the classes may become interdependent in a way that is assumed to be controlled by the user.

The paper is organized as follows. In the following section we present the basic model as a simplification and generalization of the model in [KoV92b]. Later sections introduce various extensions to this skeleton: In Section 3 we introduce a stack-like structure for keeping track of the expected right context; this is necessary for solving generally a problem arising from a certain "non-uniqueness" of syntactic atoms in typical languages and from the distributed character of the model: the parser may in some cases misunderstand the look-ahead symbol and make a wrong move. In Section 4 we show how the model can be extended with special syntactic structures, namely iteration, keyword sets, and optionality. In Section 5 we introduce an extension for storing named static program objects in an object base, and for using the classes of these objects as a resolving criteria for LL(1) parsing conflicts. This technique results in local backtracking which turns out to be useful in many practical situations. A simple syntactic error recovery technique is finally integrated with the model in Section 6. We conclude with some remarks concerning future work.

We assume that the reader is familiar with object-oriented programming. We use Eiffel [Mey88], but the techniques are in no way tied to this particular language. Except for the error recovery discussed in Section 6, the presented solutions have been tested in a slightly revised form in Eiffel.

## 2 The basic model

*Object-oriented context-free grammars*

Since we aim at an object-oriented implementation model, the application domain (in this case formal languages) must be viewed in terms of classes. This is done by interpreting each nonterminal as a class, and by regarding syntactic alternation as the basis of subclass hierarchies (i.e. each alternative form of a nonterminal gives rise to a subclass of that nonterminal). Consequently, nonterminals have to be divided into two categories: those that have a set of subclasses (and no other structural specification), and those that have a single structural specification (and no subclasses). Following the terminology of TaLE, we call the former *conceptual* nonterminals and the latter *structural* nonterminals. This kind of an object-oriented version of context-free grammars has been used by many authors, sometimes even without explicit object-orientation (for an overview, see [Kos91]). To make the distinction between conceptual and structural nonterminals more explicit, we give the production of a conceptual nonterminal in the form $A > B_1 \mid B_2 \mid ... \mid B_k$ (i.e. $B_i$, $1<i<k$, are subclasses of A), and the production of a structural nonterminal in the form $A \to B_1 B_2 ... B_k$ (i.e. $B_i$, $1<i<k$, are components of A).

*Example 1*

Consider a CFG given as follows: A -> a B, A -> C, B -> b, C -> c. This is not an OO-CFG, because A has two alternative productions that cannot be presented in the form $A > B_1 \mid B_2$. The grammar can be transfomed easily into an OO-CFG one by introducing an additional nonterminal: A > D | C, D -> aB, C -> c, B -> b. Note that a production of the form A -> B in a conventional grammar can be interpreted in two ways when viewing the grammar as an OO-CFG (assuming that A has no other alternatives): it may denote a subclass relation (B is A's subclass) or a structural definition (A consists of a single B). The choice is fairly arbitrary, but in our notation this choice is anyway made explicit (A > B or A -> B). (end of example)

Formally, we define an object-oriented context-free grammar (OO-CFG) as a 6-tuple (S,C,T,P,H,Z), where S is the set of structural nonterminal symbols, C is the set of conceptual nonterminal symbols, T is the set of terminal symbols, P is the set of productions of the form $N \to w$ ($N \in S$, $w \in (S \cup C \cup T)^*$), H is the set of hierarchy relations of the form $A > B$ ($A \in C$, $B \in (S \cup C)$), and $Z \in (S)$ is the start symbol; such that:

1) $(S \cap C = \emptyset)$ (structural and conceptual nonterminals are non-overlapping);
2) $A \in S$ implies $(A \to w) \in P$ for some w; $((A \to w) \in P$ and $(A \to w') \in P)$ 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) for each $A \in (S \cup C)$: $Z \Rightarrow^* uAv \Rightarrow^+ w$, where $u,v \in (S \cup C \cup T)^*$ and $w \in T^*$ (no useless nonterminals).

Here we assumed syntactic derivation => defined as follows: vAu => vwu, if either $(A\text{->}w)\in P$ or ($w=B$ and $(A>B)\in H$), where $A,B\in(S\cup C)$, and $v,u,w\in(S\cup C\cup T)^*$. 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 G = (S,C,T,P,H,Z) is $L(G) = \{w\in T^* \mid Z \Rightarrow^+ w\}$. In the following sections we assume that a language is generated by an OO-CFG.

Above we have generalized the definition given in [KoV92b] by allowing multiple inheritance, i.e. relations of the form A > B, C > B. Although the above definition does not rule out multiple inheritance of the same ancestor class through different paths (for instance A > B, C > B, D > A | C), this situation becomes illegal on the basis of parsing requirements - this kind of grammar is necessarily syntactically ambiguous.

*Metaobject-directed parsing*

Our aim is to represent an OO-CFG as a collection of Eiffel classes as follows:

1) there is exactly one class for each nonterminal called the *actual* nonterminal class; if A>B then the actual class of B is a subclass of the actual class of A;

2) in addition to the actual class, other auxiliary classes may be given for a nonterminal;

3) in addition to the nonterminal-specific classes, there are language-independent classes providing general implementation services.

The object representation of an input program i, obtained by instantiating the actual class of the start symbol, is a collection of the instances of the actual classes corresponding to structural nonterminals. This collection of objects is constructed as follows. Let ST(i) be the syntax tree of i, as defined by the OO-CFG. The object representation of i is OR(ST(i)), where OR is defined as:

OR(t) = nil (null object), if $root(t) \in T$;
OR(t) = OR(t'), if $root(t) \in C$ and t' is the subtree of t;
OR(t) = inst(actual(root(t)), $OR(t_1)$,..., $OR(t_k)$), if $root(t) \in S$ and $t_1,...,t_k$ are the subtrees of t.

Here root(t) gives the label associated with the root node of tree t, actual(A) gives the actual class corresponding to nonterminal A, and inst(E, $s_1,...,s_k$) gives an instance of class E with attributes referring to objects $s_1,..., s_k$. In other words, the object representation follows the structure of the syntax tree, except that conceptual nonterminals and terminal symbols are removed (conceptual nonterminals are represented by ancestor classes of the structural classes)[2].

We first note that an instance of an actual nonterminal class is created because a certain pattern has been recognized in the input program: this pattern signals that a particular actual nonterminal class should be instantiated. Since we want to distribute the information concerning nonterminals, this decision must be made in a decentralized way by an actor that is specific to a particular nonterminal. Hence we conclude that each nonterminal must have an object whose task is to create instances of the appropriate actual nonterminal class. We call this object the *metaobject* of the nonterminal, following the conventional OO terminology (see e.g. [Fer89]); the class of the metaobject is called a *metaclass*.

The metaobject of a nonterminal knows all the necessary operations and information required for creating an instance of the actual nonterminal class on the basis of input text. In particular, the metaobject has three basic operations, `look_ahead`, `match`, and `make`. `look_ahead` is a Boolean function returning true iff the corresponding nonterminal appears to be coming next in the input. This function is the essence of deterministic parsing, which is our primary aim (although in Section 5 we

[2]We will use the terms "conceptual" and "structural" freely for classes and metaobjects as well: if a nonterminal is conceptual (structural), the corresponding actual class or metaobject is likewise called conceptual (structural).

will relax this requirement a bit). Function `make` is the instantiation operation returning a new instance of the actual nonterminal class (and parsing the corresponding portion of the input while doing it).

To understand the meaning of the `match` operation, recall that only instances of structural nonterminals are created in the OO-CFG model; conceptual nonterminals are represented only as superclass layers of those instances. Hence, when it is decided that an occurrence of a particular conceptual nonterminal appears next in the input text, it is not the class of this nonterminal that should be instantiated but the class of some structural nonterminal that is a descendant of the former class. The purpose of `match` is to select and return the metaobject of that structural class, on the basis of the current input. This function is called by the `make`-operation of a conceptual nonterminal class, to decide and activate the correct metaobject.

In the case of a structural nonterminal the `make`-operation is implemented simply by calling the creation operation of the actual class. This operation will in turn ask the metaobjects of the component nonterminals to create their instances, and in that way analyze the whole input corresponding to the original structural nonterminal and construct its object representation. In the case of a conceptual nonterminal the `make`-function calls the `match`-function, and asks the returned metaobject to create the desired instance calling its `make`-function; that is, make delegates its task to another metaobject of a structural nonterminal. The overall idea of metaobject-directed analysis is depicted in Fig. 1.

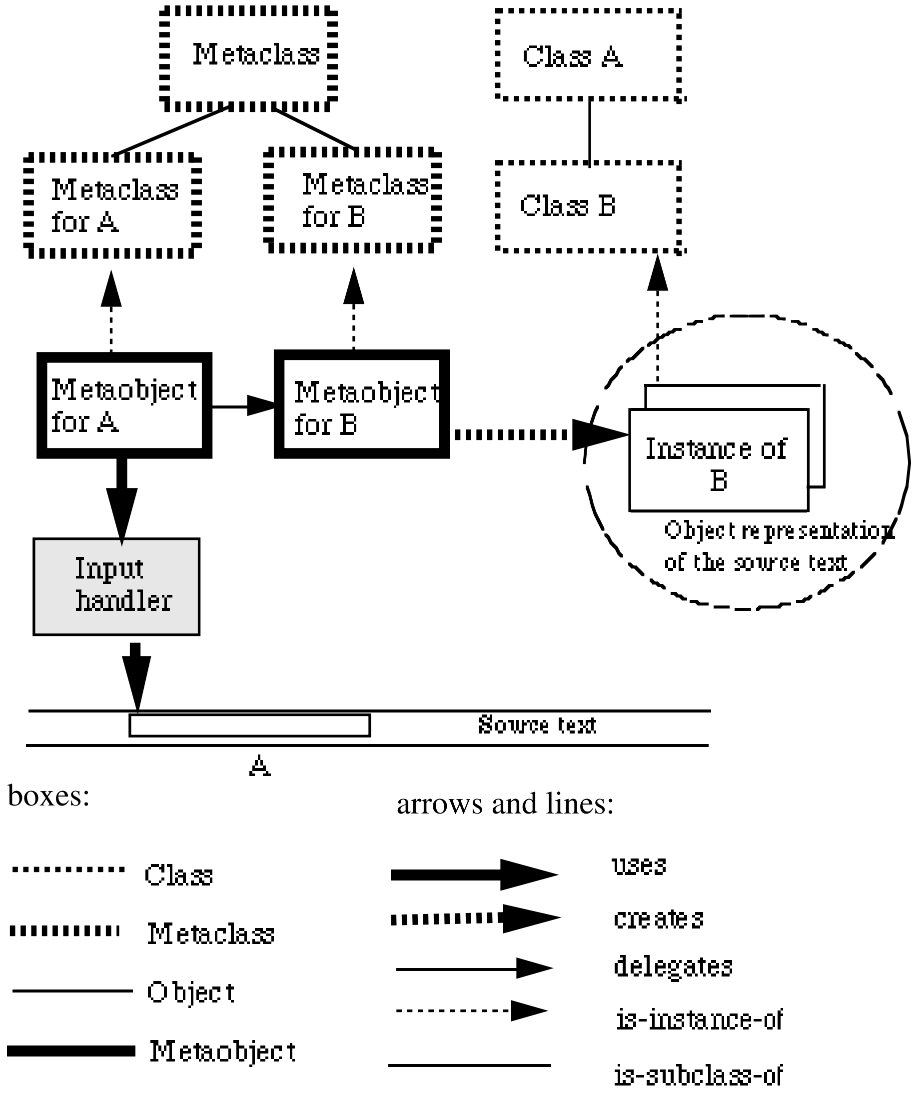


**Fig. 1**. Simplified structure of the basic implementation model. B is a structural descendant class of A.

The top-level parsing code for a structural nonterminal appears in the creation operation of the actual nonterminal class; this code resembles the traditional recursive descent code for a particular alternative production of a nonterminal. The reason for putting this code in the actual class is rather technical: it is often necessary to examine the attributes of the component structures and set the attributes of the structural nonterminal itself during its processing (i.e. during syntax analysis), and these attributes are conveniently accessible only within the actual class. Hence such "semantic" code, embedded within the parsing code, must be located in the actual classes rather than in the metaclasses. In general the idea of a metaobject would imply that all the activities associated with the instantiation of an actual class would be located in the metaclass, and in our case the parsing actions are certainly such activities. Hence it would be natural to place the parsing code entirely in metaclasses, but this would make the parse-time semantic processing technically cumbersome.

To be able to function correctly, a metaobject needs to know the starting symbols of the corresponding nonterminal. It also needs to know which structural metaobject must be activated in the case of each starter symbol. In addition, the metaobject must know whether the corresponding nonterminal can produce the empty string, and if so, through which structural metaobject. All this information must be collected dynamically to avoid dependencies between the metaclasses.

A starter symbol can be in principle any identifiable structure (keyword, token category, nonterminal) that does not produce the empty string. Since a starter symbol will be the unit which is subject to backtracking, restricting the set of allowed starter symbols is the means to regulate the degree of backtracking in our method. We will discuss this later in more detail; so far the reader may assume that starter symbols are keywords and token categories.

The starter symbol information associated with a metaobject can be represented as a list of metaobject pairs; in each pair, the first metaobject denotes the starter symbol, and the second denotes the metaobject of the structural nonterminal (class) that has to be instantiated in the case of that starter symbol. We name the components of these pairs with `starter` and `struct`. In addition, a Boolean flag `eps` indicates whether the nonterminal may produce the empty string or not, and if so, an extra attribute `eps_struct` gives the metaobject of the structural nonterminal through which the empty derivation is possible.

*Example 2*

Consider the following OO-CFG:

```
Z -> S "."                    A > F | G
S -> "begin" B "end"          F -> id G
B > S | D | E | A             G -> "skip"
D -> "decl" id                id -> ...conventional identifier...
E ->
```

The only conceptual nonterminals are A and B. This grammar will give rise to the following metaobjects and starter symbol lists. Each line describes a metaobject.

| name | eps/eps_struct | starter/struct | starter/struct | starter/struct | starter/struct |
|---|---|---|---|---|---|
| Z | no/- | "begin"/Z | | | |
| S | no/- | "begin"/S | | | |
| B | yes/E | "begin"/S | "decl"/D | "skip"/G | id/F |
| D | no/- | "decl"/D | | | |
| E | yes/E | | | | |
| A | no/- | "skip"/G | id/F | | |
| F | no/- | id/F | | | |
| G | no/- | "skip"/G | | | |
| id | no/- | id/id | | | |
| "begin" | no/- | "begin"/"begin" | | | |
| "end" | no/- | "end"/"end" | | | |
| "decl" | no/- | "decl"/"decl" | | | |
| "skip" | no/- | "skip"/"skip" | | | |

| | | |
|---|---|---|
| "." | no/- | "."/"." |

(end of example)

*Implementation of metaclasses*

Assuming that the starter lists have been constructed, the `match` function can be implemented as follows: it processes the elements of the starter list one by one, until an element is found whose `starter` metaobject returns true when its `look_ahead` function is called. Intuitively, the metaobject asks each potential starter: "Are you coming next in the input?" If no starter is found, but `eps` is true, the `eps_struct` metaobject is returned. If no starter is found and `eps` is false, an error message is given. If the starter is found, the corresponding `struct` metaobject is returned as result.

The method is highly independent on the way `look_ahead` itself is implemented: in principle any language structure can decide what is sufficient for inferring that this structure is coming next in the input. In particular, we will normally redefine this operation for each token category. Below we give a default implementation which simply tries to analyze the whole structure, and if this analysis succeeds, true is returned. In any case, the original position of the input pointer is restored. This implementation is of course always valid, but since it implies backtracking it is used only exceptionally (see below).

We assume that there are additional mechanisms which guarantee that 1) the same metaobject does not appear twice in the same list as starter component (with the exception discussed in Section 5), 2) at most one of the token metaobjects in a list returns true when being subject to call of `look_ahead`, 3) if there are two keyword metaobjects as `starter` components in a list such that one keyword is a prefix of the other, then the longer is examined first, and 4) the items with a keyword metaobject `starter` are examined before the items with a token metaobject `starter`. Note that `match` and `look_ahead` share a common part implemented below as function `screen`.

The superclass of all metaclasses, `META_NOTION`, is given in Eiffel as follows:

```
deferred class META_NOTION
   export make, look_ahead, match, eps, eps_struct, starter_list
   inherit
      SOURCE;
      EXCEPTIONS
   feature
      starter_list: SLIST[START_ITEM];
      eps: BOOLEAN;
      eps_struct: META_NOTION;

      merge(other: META_NOTION) is
      local
         sl: SLIST;
      do
         sl.Clone(other.starter_list);
         from
            sl.start
         until sl.offright
         loop
            starter_list.add(sl.item);
            sl.forth
         end;
         if other.eps then
            if eps then
               raise("Grammar error")
            else
               set_eps(other.eps_struct)
            end
         end
      end;
```

```
augment(other: META_NOTION) is
local
   it: START_ITEM
do
   from
      other.starter_list.start
   until other.starter_list.offright
   loop
      it.Clone(other.starter_list.item);
      it.set_struct(Current);
      starter_list.add(it);
      other.starter_list.forth
   end
end;

add_starter(sta: META_NOTION) is
local
   it: START_ITEM
do
   it.Create(sta, Current);
   starter_list.add(si);
end;

set_eps(m: META_NOTION) is
do
   eps:= true;
   eps_struct:= m
end;

make: NOTION is deferred end;

screen: META_NOTION is
do
   from
      starter_list.start
   until starter_list.offright or else not Result.Void
   loop
      if starter_list.item.starter.look_ahead then
         Result:= starter_list.item.struct
      end;
      starter_list.forth
   end;
end;

look_ahead: BOOLEAN is
local
   temp: NOTION;
   failure: BOOLEAN;
do
   if failure then Result:= false else
      if eps then raise("Grammar error") else
         scan.mark;
         temp:= make;
         scan.resume;
         Result:= true
      end
   end
rescue
   if is_programmer_exception("Syntax error") then
      failure:= true;
      retry
   end
end;
```

```
      match: META_NOTION is
      do
         Result:= screen;
         if Result.Void and then eps then
            Result:= eps_struct
         elsif Result.Void then
            raise("Syntax error")
         end;
      end;
   end  -- META_NOTION
```

This is a *deferred* class, meaning that it contains virtual routines (in this case only `make`) to be defined in its descendant classes. The purpose of the operations `merge` and `augment` will be explained later. Here `SOURCE` is a class providing the global source handler object called `scan`; in Eiffel this must be done by defining scan as a *once function* in `SOURCE`, creating the global object when called for the first time, and returning this object whenever called thereafter. `scan` offers various services needed by the subclasses of `META_NOTION`; we will return to these later. `EXCEPTIONS` is an Eiffel library class providing exception handling services (e.g. `raise` for raising an exception and `is_programmer_exception` for checking the name of the current exception). `NOTION` is the root class of all actual nonterminal classes. `Void` is a general function returning true iff the object reference it is applied to is nil.

In `look_ahead`, the backtracking is controlled by two operations of `scan`: `mark` stores the current input pointer in a stack, and `resume` pops the stack and changes the value of the input pointer according to the top element. We assume that the entire input is loaded into memory before the processing begins, allowing free moving of the input pointer. After the input position has been stored, the `make` operation is called, yielding the analysis of the corresponding structure. If this analysis fails, an exception will be raised (within `make`), and control is passed to the rescue part where the retry instruction reactivates the operation, returning eventually false. Hence `look_ahead` never fails to return a value.

Starter list is implemented by class `SLIST`. This is a class for list structures inheriting a library class `LINKED_LIST[START_ITEM]`, with inherited operations `start` (set cursor at the beginning of the list), `item` (the list item at the cursor position), `forth` (move the cursor forward), and `offright` (is the cursor over the right end?). In addition, `SLIST` has operation `add` that inserts a new item into its right place in the list; this operation takes care of the requirements mentioned above for a starter list. `LINKED_LIST` is a generic class parameterized with the item class; in this case `START_ITEM` is given simply:

```
class START_ITEM
   export starter, struct, set_struct
   feature
      starter: META_NOTION;
      struct:  META_NOTION;
      set_struct(str: META_NOTION) is
      do
         struct:= str
      end;
      Create(sta,str: META_NOTION) is
      do
         starter:= sta;
         struct:=  stu;
      end
end   -- START_ITEM
```

The duty of each specific metaclass (i.e. a subclass of `META_NOTION`) is to provide an implementation for the virtual `make` function (and at the same time refine the result class), and redefine - if necessary - the implementation of the `look_ahead` function. (Actually the default implementation of `look_ahead` given by `META_NOTION` will be used only for "real" nonterminals, never for keywords or token categories.)

*Metaclass for keywords*

Since keywords have trivial content and structure, it is not necessary to introduce a separate metaclass for each keyword, but the metaobjects of keywords can be created as instances of a class common to all keywords - note that this is in contrast with nonterminals and tokens having each a metaclass of their own. The actual keyword string is given as a parameter in the creation operation of the metaclass for keywords:

```
class META_KEY
   export
      repeat META_NOTION, length, rep
   inherit
      META_NOTION
         redefine look_ahead
   feature
      rep: STRING;     -- string representation of the keyword
      length: INTEGER; -- length of the keyword

      make: KEYWORD is
      do
         scan.keyword(rep)
      end;

      look_ahead: BOOLEAN is
      do
         Result:= scan.is_next(rep)
      end;

      Create(r: STRING) is
      do
         rep:= r;
         length:= r.count;
      end
   end    -- META_KEY
```

Nobody will ever consult the starter list of a keyword metaobject; hence this list need not be even initialized (although in principle we could put there an item whose both components refer to the metaobject itself, as in Example 2). Class `KEYWORD` is a dummy (no features) class needed only for technical reasons. We used here two operations of the scanner: `keyword` scans the given string in the input and moves the input pointer accordingly; `is_next` returns true iff the current input matches with the argument string (without moving the input pointer). If the current input does not match with the argument, `keyword` emits an error message. Further, the following heuristic rule is applied in `keyword`: if the argument string begins with a letter, the string must be followed in the input by a non-letter, non-digit character (this prevents the analyzer from picking up a prefix of a word as a separate keyword).

Note that the redefinition of look_ahead in META_KEY works exactly as the default implementation: the whole structure (in this case the keyword string) is analyzed without moving the input pointer. Hence the default implementation would also work, but this is a bit simpler.

*Metaclasses for token categories*

The rest of the metaclasses depend on the structural specification of the nonterminal or token category in question. As an example of a token metaclass, we could give the metaclass for a conventional identifier as follows:

```
class META_ID
   export
      repeat META_NOTION
   inherit
      META_NOTION
         redefine make,
         redefine look_ahead
   feature
      make: ID is
      do
         Result.Create
      end;

      look_ahead: BOOLEAN is
      do
         Result:= scan.next_is_letter
      end;

      Create is
      do
         starter_list.Create;
         add_starter(Current);
         eps:= false
      end
   end   -- META_ID
```

We made here use of an additional service of `scan`: `next_is_letter` returns true iff the current input character is a letter. Note that for the look-ahead of an identifier a single letter is sufficient - then a legal identifier can always be found. This demonstrates the freedom we have in implementing the look-ahead operation. Recall that the token metaobjects are consulted after the keyword metaobjects in the implementation of `match` - hence identifier-like keywords are correctly recognized.

Note that when some structural nonterminal makes use of a keyword, it knows that the component is indeed a keyword and therefore does not try to find out its "starters"; hence the starter list need not be built. In contrast, the name of the token category looks exactly the same as any nonterminal name, and therefore a token category must be treated in the same way as nonterminals by the structural nonterminal using it. For this reason the starter list of the token category must be constructed, although it is not used by the look-ahead - it can be used by the starter list construction of other nonterminals. To achieve the desired effect, a token category metaobject like the one for an identifier must insert a starter item in which both the `starter` and the `struct` components refer to the metaobject itself.

Similar metaclasses can easily be given for other token categories whose analysis and look-ahead are defined by special code. In some cases it may be necessary to implement the look-ahead in a very peculiar way; for instance, to distinguish integer and real constants in the look-ahead, the look-ahead of a real constant must find the decimal point before it can return true. We make an implicit assumption that there are never two token metaobjects in the same starter list such that their look-aheads both return true in a certain input situation - it is the language implementor's duty to make sure that this holds. To be on the safe side, the look-aheads of all tokens should be unique in every case.

In Eiffel, the metaobjects are accessed using once functions which create the objects when called for the first time, and after that simply return the identity of the existing object. This technique requires an additional class which must be inherited by every class needing access to the metaobject. Hence, for each token or nonterminal A we define class `ACCESS_A` as follows:

```
class ACCESS_A
   export metaobj_a
   feature
      metaobj_a: META_A is
      once
         Result.Create
      end
end   -- ACCESS_A
```

*Metaclasses for nonterminals*

The form of the metaclass for a nonterminal depends on the kind of the nonterminal. As an example of a conceptual nonterminal, consider the metaclass for nonterminal B in Example 2:

```
class META_B
   export repeat META_NOTION
   inherit
      META_NOTION;
      ACCESS_S;
      ACCESS_D;
      ACCESS_E;
      ACCESS_A
   feature
      make: B is
      do
         Result?= match.make
      end;
      Create is
      do
         starter_list.Create;
         merge(metaobj_s);
         merge(metaobj_d);
         merge(metaobj_e);
         merge(metaobj_a);
      end
   end   -- META_B
```

Here we used the `merge` operation: it adds copies of the elements in starter list of the argument metaobject into the starter list of the current metaobject, taking into account the requirements of starter lists mentioned previously. Further, it updates the `eps`-attributes in case the nonterminal of argument metaobject produces the empty string. See the code in `META_NOTION`. The reverse assigment attempt, denoted "?=", is a type-safe way to assign an object to a variable when the static class of the left-hand side (here `B`) is a descendant of the static class of the right-hand side (here `NOTION`): a dynamic check guarantees that the assignment is possible (here it always is). Note that the body of the creation operation consists of a call of `merge` operation for all the metaobjects of the subclasses of `B`. Since `B` (or `META_B`) must know here its subclasses statically by name, the introducing of a new subclass of `B` (or removing an existing one) necessitates the regeneration of `META_B` as well - a fact that is not quite in line with our incrementality requirements but that cannot be avoided in Eiffel (see [KoV92a]). Nevertheless, *changing* the definition of some of the subclasses does not cause regeneration of `META_B`.

For a structural nonterminal, the `make` function of the metaclass simply calls the creation operation of the actual nonterminal class, containing in turn the creation of the component structures. For instance, the metaclass of Z in Example 2 would look like:

```
class META_Z
   export
      repeat META_NOTION
   inherit
      META_NOTION;
      ACCESS_S
   feature
```

```
      make: Z is
      do
         Result.Create
      end;
      Create is
      local
         kw: META_KEY;
      do
         starter_list.Create;
         augment(metaobj_s);
         if metaobj_s.eps then
            kw.Create(".");
            add_starter(kw)
         end
      end
   end
```

In principle the creation operation "augments" the starter list with the starter lists of the metaobjects of the items in the pattern of the structural nonterminal from left to right, until an item that does not produce the empty string appears. The `augment` operation works otherwise like `merge`, but in each added starter element taken from the argument metaobject, the second component (`struct`) is changed to refer to the metaobject given as the second parameter (see the code in `META_NOTION`). If a keyword is encountered, its metaobject has to be constructed here and put in the starter list; this of course concludes the construction of the starter list because a keyword cannot produce the empty string. If there is no item that would not produce the empty string, the metaobject itself concludes that it may produce the empty string, and sets `eps` using `set_eps`.

*Actual nonterminal classes*

What remains is to show how the actual nonterminal classes are constructed. In the case of a conceptual nonterminal, the actual class is trivial: it contains nothing that is associated with syntactic analysis. It only needs to inherit its actual superclass(es), if any; if there is no actual superclass, it inherits `NOTION`.

For a structural nonterminal the actual class is more interesting. It contains the structural attributes giving the component objects, and the parsing code setting the values of these attributes. For example, consider the actual class of S in Example 2:

```
class S
   export
      repeat NOTION
   inherit
      NOTION;
      SOURCE;
      ACCESS_B
   feature
      b_component: B;
      Create is
      do
         scan.keyword("begin");
         b_component:= metaobj_b.make;
         scan.keyword("end");
      end
   end
```

Note that e.g. "`scan.keyword("begin")`" has exactly the same effect as calling the `make` function of the metaobject of "begin" - here it would only be difficult to access that metaobject. We assume that the structural attributes are suitably named, taking into account possible name clashes.

*Actual token classes*

The actual classes of token categories follow in principle the model of structural nonterminals (i.e. the creation operation is responsible for syntactic analysis of the structure), but in the case of tokens the analysis is carried out using more primitive scanning operations, and there are no internal structure (i.e. no structural attributes). For instance, an actual class could be given for id in Example 2 as follows:

```
class ID
    export
       repeat NOTION
    inherit
       NOTION;
       SOURCE;
       EXCEPTIONS
    feature
       is_id_char(c: CHARACTER): BOOLEAN is ... end;
       Create is
       do
          from
             if not scan.next_is_letter then
                raise("Syntax error")
             else
                scan.advance
             end
          until not is_id_char(scan.nextchar)
          loop
             scan.advance
          end
       end
    end     -- ID
```

Here we have again used some additional services of scan: `nextchar` returns the character at the current input position, and `advance` moves the input pointer forward with one character. An internal function is used for character classification: `is_id_char` returns true iff the argument is one of the characters allowed within this particular identifier type (say, letter, number or underscore).

The processing of an input program is started by calling `metaobj_z.make`, where Z is the start symbol of the grammar; this call returns the root object of the internal representation of the whole input program.

Note that the method is lazy in the sense that metaobjects are created only when they are really needed for a particular input. For instance, if a programmer uses only a subset of a source language implemented in this way, only metaobjects of the subset will be created. In conventional terms this would correspond to a method in which some parts of the scanner and parser are constructed dynamically, as required by the input program. This feature is important particularly when analyzing small input programs of a large language: a non-lazy but dynamic metaobject construction would make the analysis of such input programs slow.

*Tokenized nonterminals*

Above we have examined token categories whose analysis and look-ahead code is assumed to be given in a token-specific way. If a language implementation system is supposed to synthesize this code automatically, it must either know something about the nature of the token category in question, or allow some general specification method to be used by the language implementor for expressing the code. We do not want to assume a separate lexical specification method, and therefore abandon the latter approach. The former approach is applicable as far as "standard" token categories are concerned, but it should also be possible to use arbitrary token categories that can act as look-ahead symbols exactly as keywords or fixed token categories.

The obvious choice is to allow the user to define the structure of a token category in the same way as the structure of a structural nonterminal, and produce similar look-ahead and analysis code. The analysis of such tokens then becomes slightly slower (since they are parsed rather than "scanned"), but assuming that such token categories are less common this is not a serious drawback. This is also a way to associate special attributes, operations, or non-standard analysis-time actions with token categories.

To allow a structural class to act as a token category, we must take care of two things: first, the starter list of the corresponding metaobject must be constructed in a way that makes the metaobject "terminal"; second, the analysis of the input belonging to the token must be done in a special mode that prohibits the scanning of interleaving spaces. The latter requirement is easy: we can assume special operations of `scan`, say `token_mode` and `parse_mode`, which switch the mode of the scanner. To satisfy the first requirement, the starter list of the metaobject should contain a single item in which both components refer to the metaobject itself, and eps must be false. The default implementation of `look_ahead` is valid for such metaobjects: this means that during the look-ahead the whole token is analyzed (in contrast to e.g. `META_ID`). Note that the using metaobjects are not allowed to know (statically) if a particular metaobject represents a token category or not.

As an example, suppose that we "tokenize" the structural nonterminal Z whose metaclass was given previously. The "tokenized" metaclass looks like follows:

```
class META_Z_tokenized
   export
      repeat META_NOTION
   inherit
      META_NOTION;
      ACCESS_S
   feature
      make: Z is
      do
         scan.token_mode;
         Result.Create;
         scan.parse_mode;
      end;
      Create is
      do
         starter_list.Create;
         add_starter(Current);
      end
   end  -- META_Z_tokenized
```

If tokenized structural nonterminals can be nested, the mode switching operations must be implemented using an integer counter: each token_mode increments the counter and each parse_mode decrements the counter; the mode is token_mode as long as the counter is positive. Note that the possible parse-time semantic actions will be executed even if the look-ahead of a tokenized nonterminal fails; hence the language implementer should take care that such actions have no global effects.

*Optimizations*

There are several sources of inefficiency in the simplified model discussed above. First, to match the starter keywords with the current input, the metaobject should more explicitly know the string representations of the starter keywords, in a form convenient for comparing the keywords efficiently with input. This can be arranged e.g. by maintaining the keywords in a separate list, hashed according to the first character. In this way it is sufficient to examine only those keywords that have the same hash address as the next input character - usually this set of keywords contains only one element, namely the keyword that is indeed coming next.

Second, the method contains a lot of rescanning. The most obvious is the analysis of starter keywords: the look-ahead operation of a keyword actually scans the keyword (when a nonterminal metaobject is consulting its starter list), although it does not move the input pointer, and later the call of scan.keyword (re)scans the keyword and moves the pointer. This can be avoided by noting that whenever the look-ahead operation of an item succeeds during the processing of a starter list, this item will indeed be analyzed next, be it a keyword or a token. Hence, if a keyword item in a starter list succeeds in its look-ahead, the scanner should be informed that for the next activation of the `scan.keyword` operation it is sufficient only to move the input pointer forward by the length of the argument keyword, without actually analyzing the input.

Third, another type of rescanning occurs when two or more metaobjects of conceptual nonterminals examine the same input symbol for deciding the appropriate structural metaobject. Consider a grammar A -> B..., B > C | ..., C -> D..., D > E | ..., E -> "e". Assume that the input is "e". The call of `metaobj_a.make` leads to the call of `metaobj_b.make`, which in turn lets the match operation select the appropriate structural metaobject. This is done by comparing current input with B's starter keywords "e", etc. Since "e" matches, the metaobject of C gets control, and its `make` operation is activated. This will call the creation operation of C's actual class, where `metaobj_d.make` is in turn called. D's metaobject uses its `match` operation to compare current input with its starters. Again, "e" matches, and control is passed to the metaobject of E. Finally, this metaobject calls the creation operation of E's actual class where "e" is truly scanned by call `scan.keyword("e")`. Hence in this case "e" would be examined two times even if the optimization discussed above is implemented. This situation is probably less frequent, but some improvement might be achieved by eliminating the rescanning here, too. This could be done by carrying more information with the starter items. In this example the starter item "e" associated with the metaobject of C should indicate which is the nonterminal (metaobject) that caused this starter; i.e. it should have an additional pointer to the metaobject of D. Since this item is directly copied into the starter list of B, this information would be present already for the metaobject of B when it selects the structural nonterminal to be instantiated. When doing this, it could therefore inform the metaobject of D that the next time it gets control, it can directly pass control to the metaobject associated with "e" (i.e. E), without examining the input. Since this technique implies some overhead, it is not clear what would be the actual advantage; this should be investigated in practical experiments.

*Deterministic vs. backtracking analysis*

Basically we want to offer deterministic analysis: this is important for efficiency and for allowing sensible parse-time semantic actions. However, we also want to offer easy ways to circumvent the restrictions of deterministic analysis locally, and in that way solve parsing conflicts either manually or automatically. Above we already employed this for implementing arbitrary user-defined token categories acting as look-ahead symbols; in Section 5 we use this for name-analysis based look-ahead. Here we briefly discuss the relations our techniques have with general backtracking parsing.

As demonstrated above in the case of "tokenized" nonterminals, the technique allows any nonterminal to be used as a starter symbol, as long as the nonterminal does not produce the empty string. This could be generalized (assuming that no nonterminal produces the empty string): we could construct the starter list of a structural metaobject always as a single item in which both the `starter` and the `struct` components refer to the metaobject itself. Using the default implementation of `look_ahead`, the metaobject of a conceptual nonterminal would then try to analyze the leading structures of each alternative until one is found whose analysis succeeds. The structural metaobject responsible for this alternative would then get control, performing the actual parsing.

This technique implies backtracking, although not quite in the same sense as in the usual top-down backtracking parser. Hence this technique is not applicable to all context-free grammars - as general backtracking parsers - but to "generalized" LL(1) grammars in which it is required that (the grammar is given in the OO-CFG form and) for each structural choice of a nonterminal, there is a starter structure of the choice which does not produce empty and which produces a unique terminal string with respect to the starter structures of other choices. We leave the question about the general usefulness (and about the precise characterization) of this new grammar class open, but it seems certainly useful to apply this idea locally, and in that way fix the limitations of LL(1) parsing in the case of some problematic structures.

*Example 3*

As a simple example of a possible way to make use of local backtracking, consider the grammar

```
Z > X | Y
X -> A "a"
Y -> B "b"
A -> D "a"
B -> D "b"
D -> "d"
```

The grammar accepts two sentences, "d a a" and "d b b". A conventional LL(1) parser cannot handle this, because both choices begin with "d". However, if A and B are treated in the way described above, the conflict will be resolved: in the starter list of Z, the metaobject of A is the starter component yielding X, and B is the starter component yielding Y. When the look-ahead operation is executed on the metaobject of A, A will actually be parsed (assuming that the look-ahead is implemented as above), including the second input symbol. If this symbol is "a", the parse succeeds, and the input is analyzed as X. If the parse fails, the process is repeated for the second item in the starter list, which will succeed (if the input is correct); hence the input is parsed as Y.

## 3 Solving a look-ahead problem with work list

*The problem*

A central feature of the method presented above is localized scanning: there is no global scanner in the traditional sense but information about lexical atoms is distributed: each nonterminal (metaobject) knows the lexical atoms that can start the nonterminal; those lexical atoms which do not start a nonterminal appear only as actual parameters in the call of `scan.keyword`. This kind of distributed analysis causes a particular problem.

First note that if a metaobject decides that none of its starter symbols matches with the current input, the fact that it does not know all the atoms of the language does not matter - the decision is correct anyway. If it makes a positive decision, however, the situation is more complex. In the absence of global information about all the atoms of the language, this decision is guaranteed to be correct only if the lexical atoms are unique in the sense that at most one atom of the language matches with any given input situation[3].

Unfortunately, this condition is not satisfied in typical languages. In particular, there are three kinds of conflicts which create problems in this respect: 1) one keyword may be a prefix of another, 2) an instance of a token category (e.g. integer constant) may be a prefix of an instance of another token category (e.g. real constant), and 3) identifiers and keywords may have the same structure. In conventional language implementations these conflicts are solved using two basic rules: a keyword is always preferred to a token, and a longer match is preferred to a shorter one.

Our requirement is that the analyzer must handle all correct input programs correctly, and reject all incorrect input programs. With respect to this requirement, a metaobject can make a mistake when finding a match in its starter list, if a) it represents a nonterminal which can produce the empty string, and b) it confuses one of its starter symbols with the legal followers of the nonterminal in one of the three ways listed above. If so, the metaobject can make the false assumption - in the case of correct input - that there is a non-empty instance of the nonterminal, and proceed accordingly. Since in fact there was an empty instance, the analyzer will fail to analyze the input correctly. Note that if a nonterminal cannot produce the empty string, it cannot confuse its starters with its followers in the case of a correct input. If it does so in the case of incorrect input, the worst that can happen is a misleading error message.

---

[3] in fact, a slightly weaker condition would guarantee the correctness, but we will not discuss this in detail.

*Example 4*

Consider the grammar:

```
Z -> "#" S ":="
S > B | E
B -> ":"
E ->
```

Suppose the input is "# :=" (that is, S produces empty here). The problem is that the metaobject of S has no knowledge of the keyword ":="; hence when it sees that the next input character is ":" it assumes that it has found a match in its starter list, and gives control to B. Since the remaining input is "=", it cannot anymore analyze the rest of the input correctly, and reports an error (something like ":= expected").

*Example 5*

Consider the grammar

```
Z -> "b" S "e"
S > B | E
B -> id    (conventional identifier)
E ->
```

Suppose the input is "b e" (again S becomes empty). The metaobject of S recognizes identifier as a starter symbol, but has no knowledge of the keyword "e". Hence, when seeing "e" in the input it assumes there is an identifier, and decides that a non-empty S has been used - a fatal mistake.

(end of examples)

*Presentation of right context: work list*

The general idea to solve this problem is to maintain information about the right context of a nonterminal instance in the form of a *work list*. This is a list of metaobjects representing nonterminals that will be instantiated in the future, but that so far have not been seen. In top-down deterministic parsing this information is always known, although in conventional techniques it is not explicitly constructed.

The work list is constructed in a stack-like fashion as follows: Initially the metaobject of the start symbol is pushed into the list. Each time a structural nonterminal instance is created, the top element is replaced by the metaobjects of the right-hand side of the nonterminal, pushed in the reverse order. Hence the top elements in the stack indicate what is to be expected next in the input. The look-ahead problem can be solved using the work list as follows: if a metaobject decides that a non-empty alternative of a nonterminal that can produce empty must be taken, it consults the work list before accepting this decision. This is done simply by applying the look-ahead operation to each element of the work list from top to bottom, until a metaobject that does not allow empty derivation is encountered, or until a metaobject whose look-ahead overrides the original decision is encountered. If none of the look-aheads succeeds, the original decision holds. If a look-ahead on an element in the list succeeds, it competes with the original decision using the following rules:

1) a keyword always wins a token;
2) a longer keyword wins a shorter keyword.

To apply rule 2), the analyzer must know the lengths of the keywords and store them in the corresponding metaobjects. We will ignore the case in which a token competes with another token because it would imply the actual parsing of the tokens (to find out their lengths) - although this would be possible, we consider this kind of conflict so rare that is not worth the trouble. Hence we rely on the language implementer in that the look-ahead operations are sufficiently uniquely defined for token categories.

Note that this technique does not corrupt the lazyness of the analysis: the metaobjects that are pushed into the work list would be created anyway, although a bit later. The only additional object that has to be created is the work list object itself. Nevertheless, there is some overhead due to the maintaining of the list and particularly to the additional look-ahead operations applied to the list elements. In normal languages we expect this overhead to be relatively small, because maintaining a stack-like list is rather cheap, the additional look-ahead is required only for nonterminals producing the empty string, and a non-empty nonterminal is likely to appear quickly in the work list.

*Implementation of work list*

The class of the work list is defined as a subclass of a general list as follows:

```
class WORK_LIST
   export
      repeat LINKED_LIST, check_followers
   inherit
      LINKED_LIST[META_NOTION]
   feature
      wins(m1, m2: META_NOTION): BOOLEAN is
      local
         k1, k2: META_KEY
      do
         k1?= m1; k2?= m2;
         if not k1.Void then
            if not k2.Void then
               Result:= k1.length > k2.length
            else
               Result:= true
            end
         else
            Result:= false
         end
      end;   -- wins

      check_followers(orig: META_NOTION): BOOLEAN is
         -- Walk through the work list until a winner or metaobject with
          -- eps = false is encountered. A token or keyword metaobject t
         -- is a winner if wins(t,orig); a nonterminal metaobject n is a
         -- winner if wins(s,orig) for some starter metaobject in n's
         -- starter list. If a winner is found from the worklist, return
         -- true, otherwise false.
      end;  -- check_followers
   end   -- WORK_LIST
```

As usual in Eiffel, the global work list object is accessed using a once function given by an additional class:

```
class RIGHT_CONTEXT
   export
      worklist
   feature
      worklist: WORK_LIST is
      once
         Result.Create
      end
   end  -- RIGHT_CONTEXT
```

The metaclass definition has to be revised accordingly:

```
deferred class META_NOTION
   export make, look_ahead, match, eps, eps_struct, starter_list
```

```
    inherit
       SOURCE;
       EXCEPTIONS;
       RIGHT_CONTEXT
    feature
       ...
       screen: META_NOTION is
       do
          from
             starter_list.start
          until starter_list.offright or else not Result.Void
          loop
            if starter_list.item.starter.look_ahead
               and then (not eps or else
               not worklist.check_followers(starter_list.item.starter))
               then
                  Result:= starter_list.item.starter.struct
             end;
             starter_list.forth
          end;
       end;  -- screen
       ...
    end  -- META_NOTION
```

The nonterminal-specific classes have to be revised as well. The task of updating the work list is given to the metaobjects of structural nonterminals. For instance, the metaclass of Z in Example 2 becomes:

```
 class META_Z
    export
       repeat META_NOTION
    inherit
       META_NOTION;
       ACCESS_S
    feature
       meta1: META_KEY is
       once
          Result.Create(".")
       end;
       make: Z is
       do
          worklist.remove;
          worklist.put(meta1);
          worklist.put(metaobj_s);
          Result.Create
       end;
       Create is
       local
          si: START_ITEM;
       do
          starter_list.Create;
          augment(metaobj_s);
          if metaobj_s.eps then
             si.Create(meta1, Current);
             starter_list.add(si);
          end
       end
    end
```

That is, before giving control to the creation operation of the actual structural class, the `make` operation removes the top element (corresponding to Z) and inserts the metaobjects for the components of Z, in the reverse order. Note that the metaobject of keyword "." must be stored in the metaobject of Z since it must be put into the work list every time Z is entered; previously this was not necessary because such metaobjects were needed only for constructing the starter list. The metaobject is conveniently created by a once-function when it is needed for the first time.

Here we have used two additional operations on lists: `remove` deletes the list item at the cursor position, and `put` inserts a new item at the cursor position. Since the cursor position remains all the time (except for the execution of `check_followers`) at the left end of the list, the list behaves dynamically like a stack.

In the case of a keyword, the `make` operation must also remove the top element of the work list (i.e. the keyword), without adding anything:

```
class META_KEY
      ...
      make: KEYWORD is
      do
         worklist.remove;
         scan.keyword(rep)
      end;
      ...
   end    -- META_KEY
```

Similarly, the `make` operations of the metaclasses of token categories must remove the top element without adding anything (e.g. `META_ID` above).

The actual classes of structural nonterminals have to be revised, too. Since the metaobjects of the component structures can be found at the top of the work list, it is not necessary to inherit the identities of these metaobjects from the access classes - in this way the actual classes become more independent of the metaclasses. However, since the dynamic classes of the metaobjects in the work list are not known, we must use the inverse assignment attemp ("?=") instead of usual assignment - but note that the system guarantees that the class check never fails. The actual class of S in Example 2 becomes (we assume `NOTION` inherits `RIGHT_CONTEXT`):

```
class S
   export
      repeat NOTION
   inherit
      NOTION;
      SOURCE
   feature
      b_component: B;
      Create is
      local
         dummy: NOTION;
      do
         dummy:= worklist.item.make;         -- "begin"
         b_component?= worklist.item.make;   --  B
         dummy:= worklist.item.make;         -- "end"
      end
   end
```

Since the keyword objects are not needed for the internal representation, the results returned by their metaobjects are stored in a dummy variable.

The metaclasses or the actual classes of conceptual nonterminals are not changed.

# 4 Iteration, set, and optional structures

In practice it is rather cumbersome to give the syntax of a language using simple sequences of nonterminals and keywords as structural specifications. In typical languages a source program consists of various lists, like lists of statements, declarations, parameters etc. Although a list structure can be described using recursive nonterminals, the description becomes unnecessarily complicated. For this reason usual syntactic definition formalisms introduce some notation for iterative structures. In a system like TaLE lists are more or less obligatory facilities in practice.

Other special structural notations are also often useful. We will here show how the method can be easily extended to implement various list structures, optional structures, and set structures. An optional structure is one which can produce empty in addition to its actual definition. A set structure is a structural nonterminal whose structure is defined as a set of alternative keywords. Lists, optional structures, and sets are similar in that they introduce unnamed substructures that affect only the analysis of the structural nonterminal using them. This gives a hint to implement them: we create a local metaobject that represents the unnamed substructure. It is natural to create this metaobject when creating the host metaobject for the structural nonterminal, and store the metaobject in an attribute of the host metaobject. Then the unnamed substructure can be treated in exactly the same way as "normal" named structures. This demonstrates that the metaobject-directed analysis is rather flexible, because the operations of particular special metaclasses can be easily redefined without affecting the basic analysis principles.

*List structures*

In the TaLE interface, syntactic specifications are given in a graphical form containing special icons for lists, but here we use a simple textual notation: List(A,S) denotes a list of A's, separated by S's (i.e. A or ASA or ASASA or ...), where A and S can be any nonterminals, tokens or keywords. Opt(List(A,S)) denotes a possibly empty list.

Because a list introduced by a list notation is unnamed, it is not expected that semantic actions would be associated with it; a list is merely a syntactic convention. The operations that can be applied to such a list are supposed to be standard, built-in list operations (like a conceptual execution of the standard evaluation operation of all the list elements). In TaLE, there is a separate predefined class `LIST` which should not be confused with the list notation - the user may define (named) subclasses for `LIST` by giving certain semantic actions to be carried out during the processing the list, and by specifying the element and separator classes, and in that way obtain specialized list classes. An unnamed list notation, in contrast, is not a specialized subclass but only a simple list "parameterized" with the element and separator classes. The implementation model reflects this: a list notation is implemented using a standard metaclass and actual class; the creation operation of the metaclass has parameters indicating the element metaobject, the separator metaobject, and a Boolean flag indicating whether the list can be empty or not.

The standard metaclass for list notation with a nonempty separator structure is given as follows:

```
class META_ITERATION[T->NOTION]
   export
      repeat META_NOTION
   inherit
      META_NOTION;
      EXCEPTIONS
   feature
      -- iterated structure:
      body_meta: META_NOTION;
      -- separator structure:
      sep_meta: META_NOTION;

      make: ITERATION[T] is
      local
         elem: T;
         sep: NOTION;
      do
         worklist.remove;
         Result.Create;
         if look_ahead then
            from
               worklist.put(body_meta);
               elem?= body_meta.make;
```

```
            Result.add_left(elem);
         until not sep_meta.look_ahead
         loop
            worklist.put(sep_meta);
            sep:= sep_meta.make;
            worklist.put(body_meta);
            elem?= body_meta.make;
            Result.add_left(elem);
         end
      elsif not eps then
         raise("Syntax_error")
      end;
   end;

   Create(opt: BOOLEAN; body: META_NOTION;
          sep: META_NOTION) is
   do
      body_meta:= body;
      sep_meta:= sep;
      eps:= opt;
      starter_list.Create;
      augment(body);
      if body.eps then
         augment(sep);
      end;
      if sep.eps then
         raise("Grammar_error")
      end;
      if (opt and then body.eps) then
         raise("Grammar_error")
      elsif opt or else body.eps then
         set_eps(Current)
      end;
   end;
end  -- META_ITERATION
```

Because a list has a certain predefined structure, the make operation must take explicit control over the work list: before starting the analysis of each list element it inserts the metaobject of the element into the work list. After that the make operation of the element is called, which in turn replaces this metaobject by the metaobjects of its component structures. If the separator structure appears next in the input, its metaobject is put into the worklist and control is given to its make operation, after which the same is done for the metaobject of the element structure.

The creation operation constructs the starter list in the usual way. If the element structure may produce empty, the starter symbols include also the starters of the separator structure. For simplicity we make the assumption that the separator structure cannot be empty (otherwise we could not control the loop using the look-ahead of the separator).

The actual class of an iteration is simply a list that can appear as an actual class, that is:

```
class ITERATION[T->NOTION]
   export
      repeat NOTION, repeat LINKED_LIST
   inherit
      NOTION;
      LINKED_LIST[T]
   feature
      ... some standard list operations ...
   end  -- ITERATION
```

If a list is given as optional (possibly empty), the object returned by the make operation of `META_ITERATION` can be an empty list, but never `Void`.

The metaclass for a list without a separator structure is given in a similar way; we do not give it here.

A list metaobject is created by (the metaobject of) every structural nonterminal using the list notation in its syntactic specification. For instance, suppose we have A -> B Opt(List(C,D)) List(E,",") ".". The metaclass of A is given as follows:

```
class META_A
-- A = B Opt(List(C,D)) List(E,",") "."
export
    repeat META_NOTION
inherit
    META_NOTION
       redefine make;
    ACCESS_B;
    ACCESS_C;
    ACCESS_D
feature
    -- local metaobjects:
    meta1: META_ITERATION[C] is        -- Opt(List(C,D)
    once
       Result.Create(true,metaobj_c,metaobj_d)
    end;
    meta2: META_KEY is                 -- ","
    once
       Result.Create(",")
    end;
    meta3: META_ITERATION[E] is        -- List(E,",")
    once
       Result.Create(false,metaobj_e,meta2)
    end:
    meta4: META_KEY is                 -- "."
    once
       Result.Create(".")
    end;
    make: A is
    do
       worklist.remove;
       worklist.put(meta4);
       worklist.put(meta3);
       worklist.put(meta1);
       worklist.put(metaobj_b);
       Result.Create;
    end;
    Create is
    do
       starters.Create(my_class);
       augment(metaobj_b);
       if metaobj_b.eps then
          augment(meta1);
          augment(meta3);
          if meta3.eps then
             add_starter(meta4)
          end;
       end;
    end;
 end  -- META_A
```

The actual class of A treats the list notation as any other substructure: since all the metaobjects of the substructures are taken from the work list, it does not matter whether they are named or not. The creation operation of the actual class looks like:

```
Create is
local
   dummy: NOTION
do
   b_component?= worklist.item.make;
   c_list?= worklist.item.make;
   e_list?= worklist.item.make;
   dummy:= worklist.item.make
end
```

where `c_list` and `e_list` are structural attributes defined by class `A`. The names of these attributes may have to be made unique e.g. by suitable numbering; we do not discuss this problem here.

*Set structures*

A set is defined as a structure producing one of the alternative keywords; we use here the textual notation "{... alternative keywords ...}", e.g. given S -> A {",",";"} B, S can be of the form A "," B or A ";" B. A set structure is understood to have only syntactic significance, and (in contrast to lists) it will not be stored in the internal representation of the program. As for lists, in TaLE there is a predefined class for set structures (`SET`) that should not be confused with unnamed sets; named specialized set classes can be thus created as subclasses of this class. Each such subclass has its own metaclass in the usual way, whereas the unnamed set classes have the following common metaclass:

```
class META_SET
   export
      repeat META_NOTION
   inherit
      META_NOTION
   feature
      make: KEYWORD is
      do
         worklist.remove;
         from
            starter_list.start
         until not Result.Void or else offright
         loop
            if starter_list.item.starter.look_ahead then
               worklist.put(starter_list.item.starter);
               Result:= starter_list.item.starter.make
            end;
            starter_list.forth
         end
      end;
      Create is
      do
         starter_list.Create
      end
   end
```

The idea is simply to go through the starter list containing all the elements of the sets; if the look-ahead of one of the elements succeeds, the corresponding keyword metaobject is pushed into worklist, and its `make` operation is called. Recall that `KEYWORD` is only a technical dummy class. No actual class is needed for a set. In contrast to the usual case, the starter list must be effectively constructed in the metaclass of the nonterminal making use of the set notation (because the elements are known there, and it is difficult to pass them as parameters of the creation operation). For instance, consider nonterminal A with specification A -> B {";",":"}. The metaclass of A is given:

```
class META_A
   export
      repeat META_NOTION
   inherit
```

```
      META_NOTION
         redefine make;
      ACCESS_B
   feature
      meta1: META_SET is
      once
         Result.Create
      end;
      make: A is
      do
         worklist.remove;
         worklist.put(meta1);
         worklist.put(metaobj_b);
         Result.Create;
      end;
      Create is
      local
         kw: META_KEY
      do
         starter_list.Create;
         kw.Create(";");
         meta1.add_starter(kw);
         kw.Create(":");
         meta1.add_starter(kw);
         augment(metaobj_b);
         if metaobj_b.eps then
            augment(meta1)
         end
      end
   end
```

In the actual class of A the set notation is treated as any other substructure; i.e. its metaobject is obtained from the work list and called to analyze the set element. The resulting actual object is a dummy one that need not be stored permanently.

*Optional structures*

An optional structure is helpful in many practical situations, and conventional syntactic formalisms usually include some notation supporting it. As for lists and sets, TaLE provides a special graphical symbol for an optional structure; here we denote optionality by Opt(A), where A is some other single named or unnamed structure: Opt(A) denotes a structure that is either empty or equal to A. A is not allowed to produce empty itself (because then there would be two ways to produce the empty string, which would make the specification ambiguous).

We use the same strategy as before: for each occurrence of the optionality notation in the structural specifications, a metaobject is created by the metaobject of the structural nonterminal using this facility. The metaobject of the unnamed optional structure is an instance of a general metaclass for optional structures given as follows:

```
class META_OPT[T->NOTION]
   export
      repeat META_NOTION
   inherit
      META_NOTION
         redefine make
   feature
      body: META_NOTION;
      make: T is
      do
         if look_ahead then
            Result?= body.make
         else
            worklist.remove
```

```
            end
         end;
         Create(b: META_NOTION) is
         do
            body:= b;
            starter_list.Create;
            augment(body);
            set_eps(Current)
         end
      end
```

If the optional structure produces empty, `make` returns Void. There is no actual class for an optional structure because the result class of `make` is the class of the nonempty choice. Again an optional substructure is treated in the creation operation of an actual structural class in the same way as other substructures: the result returned by the `make` operation is stored in an attribute whose type is the class the nonempty choice.

# 5 Name analysis directed parsing

*Principles*

An essential part of language processing is the management and analysis of various entities named statically in the source program, like variables, constants, types, subroutines etc. Usually this part of analysis is implemented using symbol tables containing descriptors of all the static entities of the input program. Sometimes a symbol table is also used to support parsing: a parsing conflict can be solved using information found from the descriptors in the symbol table. This is a rather important feature particularly if the parsing method is somewhat restrictive, like the recursive descent method.

Perhaps the most typical case is a situation in which the parser has to make a choice between two alternative structures both beginning with an identifier. Since a normal recursive descent parser cannot make this decision on a purely syntactic basis, practical language processors consult the symbol table to find out the kind of the entity associated with the identifier appearing in the input. Usually this information is enough to make the decision. A concrete example is e.g. the analysis of assignment statements and procedure calls: since both are statements beginning with an identifier, a recursive descent parser runs into a conflict. The conflict can be solved using the symbol table, because an assignment begins with a variable identifier while a procedure call begins with a procedure identifier.

Our object-oriented implementation model offers a particularly amenable basis for developing a general technique both for maintaining "descriptors" of named entities (i.e. a symbol table) and for using this information to support parsing. This is due to the fact the objects needed for these extensions mostly already exist - we only need to reorganize them a bit. Further, the information needed by the parser is also naturally present in those objects: it is the dynamic class of the objects. We make the following basic decisions:

1) The named objects to be stored are some instances of actual nonterminal classes, belonging to the internal object representation of the program. We postulate that these objects are exactly those objects which inherit a predefined class `NAMED`. A string-valued attribute `key` of `NAMED` gives the name of the object. The collection of named objects is called the *object base*.

2) The named objects are collected into disjoint sets associated with certain instances of actual nonterminal classes, namely with those instances inheriting a predefined class `SCOPE`. A named object becomes a member in the set associated with the nearest ancestor `SCOPE` object (with respect to the tree structure of the internal program representation).

3) The analyzer can use the (dynamic) classes of the named objects as additional information to support the solving of parsing conflicts. Only those named objects that correspond to structures preceding the conflict point are taken into account.

4) Any occurrence of nonterminal or token category in the pattern of a structural nonterminal can be associated with a *qualifier*, the corresponding instance of the actual class of such nonterminal or token category is called a *denoter object*. A qualifier is the name of some descendant class of `NAMED`. If a nonterminal or token category symbol has a qualifier, it matches with the input text only if it matches syntactically *and* there is a named object (*a resolving object*) in the object base whose name is the same as the value of a dedicated attribute (`denoted_name`) of the denoter object, and whose dynamic class is a descendant of the qualifier.

The last point implies that qualifiers can be used to distinguish between two syntactic alternatives beginning in the same way, as long as there is a named object in the object base such that 1) its name is known to the denoter object, and 2) its (dynamic) class is sufficient information to resolve the conflict. This is a generalization of the usual case in which two alternative structures both begin with an identifier denoting different things: we do not demand that the qualification is associated only with an identifier but it can be associated with any structure - even with a nonterminal -, and we do not assume that the name of the resolving object is directly given by the source text, but it can be freely defined by the denoter object. The former generalization is a direct consequence of the incrementality requirement of TaLE: a structural nonterminal using some named substructure in its pattern cannot make specific assumptions about the nature of the substructure (e.g. that it denotes an identifier); the pattern must be valid even if the definition of the substructure is completely changed. The latter generalization is in turn a consequence of the first one: in the case of a qualified nonterminal it is unlikely that the name of the associated object would be the whole text generated by the nonterminal, but the name can be determined in an arbitrary way.

We use here the following textual notation for qualification: [Q]A denotes A qualified with Q. The fact that B inherits N is shown by notation $B_N$; we use this for indicating the inheriting of `NAMED` and `SCOPE`.

*Example 7*

As a simple example of the use of qualification, consider the grammar:

```
Z -> D U "."
D > A | B
A -> "a" id
B -> "b" id
U > V | W
V -> id "a"
W -> id "b"
```

We require that in V the identifier must have appeared in A, and in W the identifier must have appeared in B; i.e. the following sentences are legal: "a x x a", "b x x b"; but the following are not: "a x x b", "a x y a". This analysis problem falls beyond the range of context-free parsing, but it can be solved easily with qualification:

$Z_{SCOPE}$ -> D U "."
$D_{NAMED}$ > A | B

```
A -> "a" id           (key:= ... string of id ...)
B -> "b" id           (key:= ... string of id ...)
U > V | W
V -> [A]id "a"        (denoted_name:=  ... string of id ...)
W -> [B]id "b"        (denoted_name:=  ... string of id ...)
```

The necessary attribute assignments for the named objects (above) and for the denoter objects (below) are shown on the right. This example is a rough abstraction of the usual parsing conflict arising from two kinds of statements both beginning with an identifier.

(end of example)

Although motivated primarily by incrementality, the generalizations discussed above are actually useful for implementing certain features in programming languages. An example is the conventional dot notation which can move the critical identifier beyond the reach of the syntactic look-ahead. For instance, assume that both an assignment and a procedure call can begin with "x.y.z...". Hence it is not the leading identifier that determines the kind of the statement, but the last one (variable or procedure) which can appear arbitrarily far in the input text. In our method the whole identifier sequence can be qualified, assuming that the last identifier determines the value of the `denoted` attribute, and that the correct object can be found from the object base. The following example illustrates this.

*Example 8*

> $\text{Variable}_{\text{NAMED}}$ -> Identifier ":" Type
> $\text{Procedure}_{\text{NAMED}}$ -> "PROC" Identifier ...
> ...
> Assigment -> [Variable]Denoter ":=" Expression
> Call -> [Procedure]Denoter ActualParamPart
> Denoter -> List(Identifier,".")

(end of example)

*Revisions of language-independent classes*

Let us now consider the required new classes and revisions of the existing ones. Since any class can be qualified, the name of the resolving object must be given as an attribute of `NOTION`; we also need an attribute identifying the resolving object directly (`denoted_obj`), an operation setting this attribute (`set_denoted`), a function returning the portion of the source text corresponding to this object (`source_text`), and a function associating the current object with a named object found in the object base with the same name, given a qualifier (`attach`); the latter simply calls a function (with the same identifier) provided by the object base.

```
class NOTION
   export
      ...
   inherit
      NAMED_OBJECTS
   feature
      denoted_name: STRING;
      denoted_obj: NAMED;
      set_denoted(d: NAMED) is
      do
         denoted_obj:= d
      end;
      source_text: STRING is
      do
         Result:= ... the source text corresponding to this instance ...
      end;
      attach(qua: META_NOTION): BOOLEAN is
      do
         Result:= objectbase.attach(Current,qua)
      end;
      ...
```

The collection of named objects is organized as a tree of `SCOPE` objects, each associated with a set of `NAMED` objects. It should be emphasized that the `NAMED` and `SCOPE` objects are the same objects appearing in the internal representation of the input program - the tree actually contains only references to these objects. We assume that one of the `SCOPE` objects in the tree is *current* at any one time. This global tree is accessible in the usual way through a once function:

```
class NAMED_OBJECTS
   export objectbase
   feature
      objectbase: OBJECT_BASE is
      once
         Result.Create
      end
   end
```

Class `OBJECT_BASE` is given below. Since the exact implementation of the operations `look_up` and `insert` in this class may depend (and therefore will be redefined) on the language to be implemented, we will explain them only informally[4].

```
class OBJECT_BASE
export
   look_up, insert, attach,
   enter_scope, exit_scope
inherit
   COMPACT_TREE[SCOPE];
   EXCEPTIONS
feature
   look_up(obj_name: STRING): NAMED is
      -- return the named object whose key attribute equals to
      -- obj_name; if none exists, return Void
   end;

   attach(arg_obj: NOTION; qualifier: META_NOTION): BOOLEAN is
   local
      n: NAMED
   do
      n:= look_up(arg_obj.denoted_name);
      if not n.Void then
         if not n.creator.compatible(qualifier) then
            Result:= false
         else
            arg_obj.set_denoted(n);
            Result:= true;
         end;
      end;
   end;   -- attach

   insert(x: NAMED): BOOLEAN is
      -- insert x into the current scope of the object base;
      -- if an object with the same name already exists in the
      -- scope, return false, otherwise true
   end;

   enter_scope(s: SCOPE) is
   do
      -- insert a new scope node s into the object base as the
      -- son of the current scope, and let the new scope
      -- be the current scope
   end;

   exit_scope is
   do
     -- let the parent node of the current scope be the new
      -- current scope
   end;
```

[4]In fact, it would be likely sensible to define a somewhat richer collection of operations for `OBJECT_BASE`, allowing the implementation of various non-trivial scope rules; we are here contented with simple lookup and insert routines, and scope entrance and exit routines. The implementation of non-trivial scope rules falls beyond the subject of this paper.

```
end -- OBJECT_BASE
```

Function `attach` makes the association with the denoter object and the corresponding named object, using function `look_up` to locate the named object in the object base. If the association is successful, the function returns true, otherwise false. The function makes use of a new function `compatible` defined for all metaobjects; this function returns true iff the actual class of the nonterminal represented by the current metaobject (executing the function) is a descendant of the actual class of the nonterminal represented by the metaobject given as parameter.

Class `NAMED` defines the string-valued `key` attribute. Further, to allow the checking of class compatibility, a named object must know the metaobject that created it: this metaobject represents the dynamic class of the named object. The creating metaobject is given by attribute `creator`. Operations are needed for setting creator and for inserting the object into object base:

```
class NAMED
   export key, creator, set_creator
   inherit
      NAMED_OBJECTS;
      ERROR_MSG
   feature
      key: STRING;
      creator: META_NOTION;
      set_creator(m: META_NOTION) is
      do
         creator:= m
      end;
      insert is
      do
         if key.Void then
            error("Naming error")
         elsif not objectbase.insert(Current) then
            error("Naming error")
         end
      end;
   end  -- NAMED
```

Class `SCOPE` defines an attribute (`obj_set`) that gives the associated set of named objects. Further, the class defines scope operations, namely the operations for entering and exiting a scope; these are implemented directly using the corresponding operations of `OBJECT_BASE`.

```
class SCOPE
   export obj_set, enter_scope, exit_scope
   inherit
      NAMED_OBJECTS
   feature
      obj_set: OBJECT_SET;
      enter_scope is
      do
         object_base.enter_scope(Current);
      end;
      exit_scope is
      do
         object_base.exit_scope;
      end;
   end  -- SCOPE
```

Class `OBJECT_SET` describes a set of named objects associarted with the scope. It defines operations for inserting a new object into the set, and for finding an object with a given name in the set; these operations are employed by the lookup and insert operations of class `OBJECT_BASE`. The class is implemented as a hashed structure; we will not present this class here in detail.

We also have to revise the structure of the starter lists by introducing an additional component in each starter item; this component gives the qualifier (metaobject) of the associated starter symbol, if any. Hence, we have:

```
class START_ITEM
   export starter, struct, set_struct, qualifier
   feature
      starter: META_NOTION;
      struct:  META_NOTION;
      qualifier: META_NOTION;                --  <---
      set_struct(str: META_NOTION) is
      do
         struct:= str
      end;
      Create(sta,str,qua: META_NOTION) is
      do
         starter:= sta;
         struct:=  stu;
         qualifier:= qua                    --  <---
      end
end   -- START_ITEM
```

Class `META_NOTION` needs some revisions, too. Function `compatible` has to be defined as a deferred routine, to be defined separately by each actual metaclass (see below). The `make` operation has to be augmented with a parameter giving the possible qualifier (as a metaobject): note that in different contexts the same structure symbol may be associated with different qualifiers or with no qualifier at all. Finally, the implementation of the `screen` operation has to changed; this is actually the core of the technique:

```
deferred class META_NOTION
   export
      make, look_ahead, match, eps, eps_struct, starter_list,
      compatible
   ...
   feature
      ...
      compatible(other: META_NOTION): BOOLEAN is deferred end;
      ...
      make(qua: META_NOTION): NOTION is deferred end;
      ...
      screen: META_NOTION is
      local
         candidate: START_ITEM
      do
         from
            if not failure then
               starter_list.start
            else
               starter_list.forth
            end
         until starter_list.offright or else not Result.Void
         loop
            candidate:= starter_list.item;
            if candidate.starter.look_ahead then
               if candidate.qualifier.Void then
                  Result:= candidate.struct
               else
                  scan.mark;
                  temp:= candidate.starter.make(candidate.qualifier);
                  scan.resume;
                  if not temp.Void then
                     Result:= candidate.struct;
                  end
               end
            end;
```

```
            starter_list.forth
         end;
      rescue
         failure:= true;
         scan.resume;
         retry
      end;  -- screen

      add_starter(sta, qua: META_NOTION) is
      local
         it: START_ITEM
      do
         it.Create(sta,Current,qua);
         starter_list.add(it)
      end;  -- add_starter
      ...
   end   -- META_NOTION
```

The new implementation of `screen` operation needs some explanations. If there are no qualified items in the starter list, the operation works exactly as before. However, if a starter item is qualified and its look-ahead succeeds, the starter item has to be actually parsed so that the name analysis can be made. Hence in that case we call the make operation of the starter metaobject (in the case of an identifier starter, which is presumably the usual case, this means that the identifier is scanned). If the parse was successful and the corresponding object was found in the object base, the resulting instance is stored in `temp`. If the parse succeeds but there is no corresponding metaobject, `temp` becomes Void, the state of the input pointer is restored, and the processing of the starter list continues. If the parse fails, an exception is raised within `make`, and the exception is caught by the rescue clause where the input pointer is again restored so that the processing of the starter list may continue. Note that in all cases the `screen` operation does not change the state of the input pointer. Also note that the calls of screen may be recursively nested, but for a particular call of screen, mark and resume are always called pairwise, guaranteeing that the desired state of the input pointer is obtained.

*Revision of language-dependent classes*

The basic machinery for implementing name analysis and name-dependent syntactic analysis is now completed; the remaining task is to revise the language-specific classes so that they make use of this machinery. We retain the principle that the duty of a metaobject is to take care of all the preparatory actions associated with the creation of an instance belonging to the internal representation of the input program. These actions include the maintaining of the object base; hence the metaclass construction for individual nonterminals must be revised

In the creation operation of a metaclass, the starter list is constructed as before, but qualified symbols are treated as primitive symbols: they cause a single starter item like keywords. For simplicity, we assume here that qualified structures cannot be empty; hence the search for possible starter symbols need not proceed behind a qualified symbol. Actually we could easily relax this requirement, but since it seems that qualified structures producing empty are extremely rare in practice, we do not consider this restriction essential.

The metaclass of a conceptual nonterminal remains the same; recall that the role of a conceptual metaobject is merely to dispatch the call for a `make` operation to the appropriate structural metaobject. The qualifier parameter must be added:

```
make(qua: META_NOTION): A is
do
   Result?= match.make(qua)
end;
```

Each metaclass must also provide an implementation for `compatible`. For nonterminal A, the metaclass `META_A` is given as follows:

```
compatible(other: META_NOTION): BOOLEAN is
do
   Result:= other = Current or else
            metaobj_a1.compatible(other) or else
            ...
            metaobj_ak.compatible(other)
end
```

where A1, ... , Ak are those immediate superclasses of A that are descendants of `NAMED`.

A structural metaclass depends on whether the actual nonterminal class inherits `NAMED` or `SCOPE`. If it inherits neither of them, the `make` and creation operations are slightly revised. For an example of the effect of qualification, consider the metaclass for A -> [C]B D:

```
class META_A
   export
      repeat META_NOTION
   inherit
      META_NOTION;
      ACCESS_B;
      ACCESS_C;
      ACCESS_D
   feature
      make(qua: META_NOTION): A is
      do
         worklist.remove;
         worklist.put(metaobj_d);
         worklist.put(metaobj_b);
         Result.Create;
         if not qua.Void then
            if not Result.attach(qua) then
               error("Naming error");
               Result.Forget
            end
         end
      end;
      Create is
      local
         si: START_ITEM
      do
         starter_list.Create;
         add_starter(metaobj_b,metaobj_c);
      end
   end
```

Note that D is not consulted for finding new starters, the qualified B terminates the process as if it were a keyword. The actual class of A would be as follows:

```
class A
   export
      repeat ...
   inherit
      ...
   feature
      b_component: B;
      d_component: D;
      Create is
      do
         b_component?= worklist.item.make(metaobj_c);
         d_component?= worklist.item.make(nil);
      end
   end   -- A
```

Here `nil` is assumed to denote the null reference (surprisingly, in Eiffel 2.3 there is no such explicit value, but `nil` can be realized e.g. as an uninitialized attribute).

For a structural descendant class of `NAMED`, the metaobject must insert the object into the object base, and set the value of its `creator` attribute; this can be done immediately after creating the object. The value of the `key` attribute of the named object is assumed to be set previously during the processing of the named structure.

```
make(qua: META_NOTION): N is
do
    ...
    Result.Create;
    Result.set_creator(Current);
    Result.insert;
    if not qua.Void then
        if not Result.attach(qua) then
            error("Naming error");
            Result.Forget
        end
    end
end;
```

For a structural descendant class of `SCOPE`, the metaobject must take care of exiting the scope after processing the scope structure. However, the entering operation cannot be executed by the metaobject because the operation should be done prior to processing the structure, and the scope object does not exist before calling the Create operation. Hence this operation must be called as the first statement in the creation operation of the actual class, where the scope object is available as the current object.

*Optimizations*

The name analysis technique presented above contains an obvious source of inefficiency: each object base look-up that originates from a qualified item appearing as a starter symbol is performed at least twice, once for look-ahead and once for actual parsing. In some cases the object base look-up may have to be repeated even more than once. For instance, consider the nonterminals A > B | C, B -> D..., D > E | F, E -> [F]G ... . In the starter list of A, [F]G is first used to distinguish between B and C, consulting the object base. If B is selected, control is given to the metaobject of D which again consults the object base. Finally, during the parsing of E, [F]G is eventually analyzed, and the object base is consulted for the third time. Although this is perhaps not a typical case in practical languages, the backtracking included in the use of the object base implies a significant potential overhead.

Using the optimization technique discussed in Section 2, the repeating of the object base look-up can be reduced to one (i.e., successive look-aheads need not re-examine the starter list). The remaining problem is how to eliminate the reparsing (including the object base look-up) during actual parsing phase, when this analysis has already been carried out during the look-ahead phase. We can do this e.g. as follows: each metaobject saves a reference to the actual nonterminal instance it has created last. Whenever a qualified item is preparsed successfully during look-ahead, the corresponding metaobject is "passivated" by setting a flag attribute. If a metaobject receives a request to create an actual nonterminal instance through `make` in the passive state, it (resets the flag and) returns the instance created last, without performing any further analysis.

There is also "horizontal" reprocessing involved in the method: if a starter list contains occurrences of the same starter item but with different qualifiers (say, a variable identifier and a procedure identifier), the starter item may be processed several times and the same key may be used for several object base look-ups, depending on the order in which the starter list is processed. Note that above we discussed how a *successful* object base look-up need not be repeated, but here the problem is that even though a perfect match is not found for a qualified item, its syntactic form may nevertheless be already analyzed, and the corresponding object base element may have been found (albeit with a wrong qualifier). These are analysis results that should be retained if possible. We will not discuss the necessary techniques here in more detail.

# 6 Error recovery

Syntactic error recovery has lost some of its significance as a result of development in programming languages and programming techniques. This is mainly due to the fact that compiled program units (modules, classes) have become smaller and smaller; typically, a separately compiled unit includes less than a hundred lines of code. The situation was essentially different, say, fifteen years ago when programs of tens of thousands of lines were compiled in a single batch: then error recovery was an absolute necessity for reducing the number of compilation cycles.

Since the nature of the languages implemented through TaLE is not known, we want to include at least a simple error recovery technique in the produced language processors. The particular problem in our case is that in our distributed model a) the follower symbols are not explicitly known, and b) the complete set of terminal symbols is not explicitly known. Since this information is used by the traditional methods, we have to either collect this information dynamically or use some non-traditional method. In contrast to the original plan [JKP91], we have decided to take the latter approach: collecting the necessary information at run-time seems to be overly complicated and time-consuming, with respect to the significance of error recovery.

We aim at a simple, heuristic technique that works reasonably well in the majority of practical cases. The basic principle is that after an error the processing continues from the next major syntactic unit that is probably not affected by the error.

We make the assumption that major syntactic units are elements of lists. This is rather obvious; in fact it is very difficult to define a sensible language without using the list structure. In most languages programs consist of lists of major structures. Since the elements of lists are syntactically independent, a promising strategy is to discard the current (erroneous) list element, and skip the input until the beginning of the next element in the current list structure (or until the end of the list, if there is no next element). If the list element is sizable, some later errors may remain undetected, but this can anyway happen in all methods.

The remaining problem is how to find the next element in the list. Since our analyzer has no knowledge of the complete set of terminal symbols, it is difficult to skip the input in a controlled way to find a particular starter symbol: there will probably be many symbols in between that are completely unknown even to the metaobjects in the worklist. For instance, if a list element starts with an identifier, the analyzer can easily confuse it with a keyword of an unknown structure. However, a suitable landmark is provided by the separator structure: usually it consists of a single, unique special symbol (like a semicolon) that is easy to recognize in the input. Even if the separator structure allows several keywords (e.g. a set structure), the recognition is relatively safe. Only if the separator structure can begin with an identifier, or there is no separator structure at all, the finding of the next list element becomes problematic. Rather than trying to develop sophisticated techniques handling these cases as well, we use a keyword separator as a basis of the method, and apologize for less brilliant recovery in some (hopefully rare) cases.

Since error recovery is associated with lists in our method, most of the required activities will be on the responsibility of the metaobject for an iteration. We revise class `META_ITERATION` as follows:

```
class META_ITERATION[T->NOTION]
   ...
   feature
      -- iterated structure:
      body_meta: META_NOTION;
      -- separator structure:
      sep_meta: META_NOTION;

      make: ITERATION[T] is
      local
         elem: T;
         sep: NOTION;
         failure: BOOLEAN;
         wl_mark: INTEGER;
      do
```

```
      if not finish then
         if not failure then
            worklist.remove;
            wl_mark:= worklist.mark;
            Result.Create;
         end;
         if look_ahead then
            from
               worklist.put(body_meta);
               elem?= body_meta.make;
               Result.add_left(elem);
            until not sep_meta.look_ahead
            loop
               worklist.put(sep_meta);
               sep:= sep_meta.make;
               worklist.put(body_meta);
               elem?= body_meta.make;
               Result.add_left(elem);
            end
         elsif not eps then
            raise("Syntax_error")
         end
      end
   rescue
      if is_programmer_exception("Syntax error") then
         failure:= true;
         finish:= scan.skip(sep_meta);
         worklist.resume(wl_mark);
         retry
      end
   end;
   ...
end  -- META_ITERATION
```

Here we used two additional operations of the work list: we assume that the cursor position of the current "top" element is returned by `mark`, and that `resume` restores the top cursor position given as parameter (thereby removing elements "above" the given point). The new `skip` function of the scanner moves the input pointer forward until the look-ahead operation of the separator metaobject returns true, or until a follower of the list structure (given by the worklist) is encountered in the input. In the former case `skip` returns false and moves the input pointer over the separator symbol, in the latter case the next input item will be the follower symbol and true is returned. If neither a separator nor a follower symbol is found, skip will read the input until the end of file, raise another exception ("Input error"), and return true.

# 7 Concluding remarks

A basic motivation of this research has been to study the applicability of object-oriented techniques in language implementation: although there are many object-oriented language implementation systems, the models for implementation expressed at the level of a general OO programming language are less known. Since such models can be used both in hand-written and automated language implementation, they are perhaps even more significant than individual language implementation systems.

Our work demonstrates that object-oriented techniques are well suited to language implementation. In particular, the following advantages of using OO techniques could be observed:

1) OO techniques make it possible to distribute the knowledge of the source language in independent classes representing the natural units of the language. In this way a language implementation can be developed incrementally, which is a clear advantage both in hand-written and automated implementation.

2) OO techniques allow a simple, unified model of language processing. This is due to the fact that all entities are treated basically in the same way as objects. The model is intuitively clear because it is structured according to the high-level concepts of the source language rather than according to certain implementation-oriented concerns like scanning, parsing, semantic analysis etc.

3) A nonterminal can have a natural run-time representation, namely a metaobject. In this way a nonterminal, with all its capabilities and information, can be freely manipulated as data, stored in various data structures, and activated to provide services like look-ahead or analysis. The same metaobject can thus serve many purposes.

4) We have made extensive use of the concept of a once function in Eiffel. This feature has turned out to be surprisingly useful for expressing lazy parser construction and other activities that must be postponed until really needed.

5) Dynamic binding makes it possible to specialize operations associated with nonterminals in a very flexible way, without sacrificing the unified, general model. The basic model is essentially centered around two operations, one for look-ahead and one for analysis, and the basic model is highly insensitive to the way these operations are implemented for individual nonterminals. This property was particularly useful for obtaining restricted backtracking which was essential for a general technique for integrating name analysis and parsing.

6) The internal (object) representation of the source program is directly implied by the nonterminal classes, and is a natural by-product of analysis. This representation is more abstract than a conventional parse tree because the chain productions originating from syntatic alternation are eliminated. The objects in the internal representation are hierarchically classified, and they can be naturally associated with various operations to perform semantic processing (like interpretation or translation).

An obvious topic for future research is the optimization of the techniques presented here. In particular, the metaobject-directed parsing method contains many sources of inefficiency in its basic form. Since the use of an object-oriented language in itself implies certain run-time over-head, it is difficult to estimate how near to the conventional efficiency we can get. Nevertheless, it seems that inefficiency will remain to be the main (albeit in many cases not very serious) drawback of using an object-oriented model in language implementation. Current work concentrates on experiments for measuring the time consumption of the proposed language analysis techniques, and for developing various optimizations of the basic method.

**Acknowledgements**

This work is supported by the Academy of Finland through grant 1061120.

# References

[AlR92] Alpert S.R., Rosson M.B.: ParCE: An Object-Oriented Approach to Context-Free Parsing. Comput. Syst. Sci. Eng. 7, 2 (April 1992), 136-144.

[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.

[Gra92] Graver J.O.: The Evolution of an Object-Oriented Compiler Framework. Software Practice & Experience 22, 7 (July 1992), 519-535.

[Gro90] Grosch J.: Object-Oriented Attribute Grammars. In: Proc. 5th International Symposium on Computer and Information Sciences (ISCIS V), A.E. Harmanci, E. Gelenbe (eds.), Cappadocia, Nevsehir, Turkey, 1990, 807-816.

[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.

[Hed92] Hedin G.: Incremental Semantic Analysis. Ph. D. thesis, Department of Computer Science, Lund University, March 1992.

[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.

[JäK93] Järnvall E., Koskimies K.: Object-Oriented Language Engineering: The TaLE Approach. Unpublished manuscript, 1993.

[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.

[Kos90] Koskimies K.: Lazy Recursive Descent Parsing for Modular Language Implementation. Software Practice & Experience 20,8 (August 1990), 749-772.

[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.

[KoV92a] Koskimies K., Vihavainen J.: The Problem of Unexpected Subclasses. Journal of Object-Oriented Programming, October 1992, 25-31.

[Kov92b] Koskimies K., Vihavainen J.: Incremental Parser Comnstruction with Metaobjects. Report A-1992-5, Department of Computer Science, University of Tampere, November 1992.

[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. of TOOLS '90, Paris 1990.

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