# Techniques for Modular Language Implementation*

KAI KOSKIMIES

*Department of Computer Science, University of Tampere*
*Box 607, SF—33101 Tampere, Finland*
e-mail: koskimie@ondake.uta.fi

## Abstract

It is argued that the modularization of language implementation software should be based on the concepts of the source language rather than on certain implementation techniques: this would lead to more maintainable and reusable software components. Various techniques supporting source language oriented modularization are explored, covering both syntactic and semantic issues. For scanning and parsing, a lazy LL(1) method based on independent nonterminal modules is proposed; in this method the scanner and the parser are partially constructed during parsing according to the needs of a particular input. For semantic aspects, an object-oriented approach is suggested in which the source program is viewed as a collection of objects. The classes are derived systematically on the basis of a disciplined syntactic specification of the language.

## 1. Introduction

A crucial question of any software development is how to divide the software into managable pieces, modules, with simple mutual relationships. The answer can vary considerably, depending on the way a system designer thinks about the system. There are at least two basic approaches. In the implementation-oriented approach the system is viewed as a hierarchy of abstract machines; then the modules provide services required by the abstract machines. In the task-oriented approach the system is divided into pieces according to the logical task of the system, so that different modules implement different subtasks. An important advantage of the latter approach is that if the task is slightly changed, the system can be relatively easily updated by

* Lecture presented at the 1st Finnish-Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

replacing the corresponding modules with new ones. The latter view is normally taken also in object-oriented programming.

As a software product, a language system (analyzer, compiler, translator, interpreter) is perhaps one of the most studied. The structure of such a system has become practically standard, and the components can be usually developed using well-known systematic techniques, often supported by automatic generation tools. The standard structure of a language system follows basically the implementation-oriented approach: typical modules are an input buffer, a scanner, symbol table, code generation services (see e.g. [WeM 80]). These modules can be understood as an abstract implementation machine for a particular language.

It is somewhat surprising that alternative modularization techniques, in particular the task-oriented approach, have not been applied in practical language implementation. The task-oriented approach (which can be called *language-oriented* approach in the context of language systems) has obvious advantages over the implementation-oriented approach:

- the number of modules depends on the size of the language, implying that the sizes of the modules remain small;
- during language development, some part of the language can be easily changed by replacing the module corresponding to that part with another module;
- components of existing language systems can be reused in the development of new languages;
- system maintenance becomes easier because of fine-grained modularization that can be understood on a high conceptual level (i.e., on the level of the source language).

Although the language-oriented modularization principle has not been applied in practical implementations of programming languages (to my knowledge), it is not a completely new idea in the research. From a theoretical point of view, the subject has been studied by Watt [Wat 85]. Some experimental language implementation systems provide a modular specification language (e.g. [Toc 88]). In some implementation systems ([Gro 84], [HeR 75]) a language implementation can be developed in a step-wise way that is ideologically close to the language-oriented modularization.

In this paper we study the language-oriented modularization on the level of a general-purpose modular implementation language (say, Modula-2 or Oberon [Wir 88]). Our results can be applied to writing modular language systems by hand, but they can equally well be used in the design of a generator producing (modular) implementations on the basis of high-level specifications. We feel that even in a system providing a modular specification language the generated code should also be modular: otherwise a small change in some of the specification modules requires a complete recompilation of the generated code (even though the other specification modules perhaps need not be reprocessed).

We proceed as follows. In the next section we introduce a notation for describing the construction of programs; we will use this notation throughout the paper. Section 3 is an informal introduction to a parsing technique supporting modular implementations; this part is essentially a summary of the results given in [Kos 89]. In

Section 4 we present a method for constructing an abstract representation of a program in a modular way. Section 5 discusses briefly the problems in modularizing the dynamic semantics. Sections 4 and 5 are mostly extensions of the ideas presented in [Kos 88]. Finally, in Section 6 we present some concluding remarks.

## 2. Program generation tools

We make use of static statements enclosed in brackets; these statements can be regarded as advanced "macro" facilities that can be used within normal program text. They are assumed to be executed by a preprocessor (or by the compiler) to produce the actual source code to be inserted in their places. Hence, the information on which the execution depends must be static. We use three kinds of static statements. The ***let*** statement allows the value of a static expression to be inserted in the code:

$$[\textit{let}\ X = E{:}\beta]$$

where $E$ is a (static expression) and $\beta$ is an arbitrary string. As a result, the string obtained from $\beta$ by replacing every occurrence of $X$ with the string representation of the value of $E$ is inserted into the source code at this point. A short-hand notation can be used for nested ***let*** statements:

$$[\textit{let}\ X_1 = E_1{:}[\textit{let}\ X_2 = E_2{:}\ \ldots{:}\beta]]$$

can be written in the short form:

$$[\textit{let}\ X_1 = E_1, X_2 = E_2, \ldots{:}\beta].$$

Static ***if*** statement is given in the form:

$$[\textit{if}\ E{:}\beta]$$

where $E$ is a condition (Boolean expression) and $\beta$ is an arbitrary string. The condition must be a static expression; if it yields true $\beta$ is included in the program; otherwise the entire statement is ignored by the compiler. Similarly, a static ***for*** statement

$$[\textit{for}\ X\ \textit{in}\ S{:}\beta]$$

denotes a sequence of strings, each obtained from $\beta$ by replacing the occurrences of $X$ with one element in the ordered set $S$. The above statement then generates:

$$\beta_1\beta_2 \ldots \beta_k$$

where $\beta_i$ is obtained from $\beta$ by replacing every occurrence of $X$ with the $i$th element of $S$. We assume that all sets discussed here are ordered; if the order is not explicitly given some arbitrary order is assumed. Static statements may be nested, in which case the outermost statements are executed first. Note that we use italic bold for the keywords of static statements to distinguish them from the keywords of the normal program text.

We use these static statements mainly to express the generation of programs in a compact way: a program containing static statements can be understood as an algorithm for producing a normal program.

## 3. Scanning and parsing

We define first some basic concepts. A context-free grammar (CFG) is a 4-tuple $(V_T, V_N, S, P)$, where $V_T$ is the set of terminal symbols, $V_N$ is the set of nonterminal symbols, $S$ is the start symbol and $P$ is the set of productions of the form $A \rightarrow \beta$, where $A$ is a nonterminal and $\beta$ is a possibly empty string of terminals and nonterminals. A production of the form $A \rightarrow B$, where $B$ is a nonterminal, is called a *chain production*. A CFG is *reduced* if every nonterminal is used in the derivation of a terminal string. A CFG in *non-circular* if there is no nonterminal that can produce a string consisting of this nonterminal only.

Assuming that the modularization is based on language concepts that are (mostly) represented by certain syntactic structures, we decide that for each nonterminal of the language there is a separate module that implements this nonterminal. It might be argued that this decision leads to a huge number of modules in some cases (say, several hundreds), but on the other hand it allows very fine-grained reuse of language structures. We do not regard the possibly great number of modules as a serious problem, assuming that the module library is organized in some sensible way.

The natural way to proceed is then to introduce a handling procedure for each nonterminal module, taking care of the processing of the structure generated by that nonterminal in the well-known recursive descent style. However, the basic modularization principle requires that when writing one module we are not allowed to make use of the detailed knowledge of the tasks of the other modules. This principle guarantees that the modules are interchangeable, as long as the interfaces remain the same. When applied to nonterminal modules, this means that we must be able to replace the implementation part of a nonterminal module into another one without affecting the implementation of the other modules. In language terms, if we change the productions of a nonterminal, it must be sufficient to change the implementation part of that nonterminal only. Note that if the processing procedures are written in the traditional way, this does not hold because the starter and follower symbols of nonterminals are assumed to be known globally, and because all the terminal tokens of the language are assumed to be known by a scanner.

Hence, our problem is the following: how can we write the analysis procedure for one nonterminal module on the basis of the productions of that nonterminal only, without using any knowledge about the productions of other nonterminals and the tokens appearing in them? This implies that the global information about the entire language cannot be embedded statically into the program code, but it must be computed at run-time. The key question is when and how to collect this information. In a nondeterministic top-down analyzer (e.g. [Gro 84], [HeR 75]) the necessary information is essentially recomputed every time it is needed. The other extreme is to compute all the information before the analysis of an input begins. In both cases some loss of efficiency is expected: the former method involves backtracking (which is unpleasent also for the semantic processing), the latter method implies that the analysis time of every program is increased by the time required for parser and scanner construction which is particularly unsatisfactory for small programs of a large language.

Our choice is a method which is between these two extremes. We find out the necessary information about the grammar on the fly during parsing, and store it so that it need not be recomputed when the same parsing situation occurs later. This

means that we construct the parser during parsing, but only as far as is needed to analyze a particular input text. This approach can be called *lazy* in the sense that the analyzer is constructed in a lazy manner. The lazy approach has been previously taken in the context of LR parsing by Heering, Klint and Rekers in [HKR 88].

In a traditional recursive descent parser, global grammar information is used only to select the alternative production of a nonterminal, when the procedure of the nonterminal has been activated. Hence, this part of an analyzer procedure must be removed so that the selection can be based on some global data structure that is built during parsing. The analyzer will therefore be partly table-driven (the global data structure for selecting the alternative), partly hard-coded (the code for analyzing the right-hand sides of productions).

Obviously it is possible to give each nonterminal module a procedure that computes the starter symbols of that nonterminal (say $A$), using the corresponding procedures of those nonterminals appearing on the left-hand sides of the productions of $A$. Then a straightforward way to construct a lazy recursive descent parser would be to augment each analyzer procedure with an initial action that computes and stores the starter symbols of that nonterminal, together with information that indicates which production must be selected for each starter symbol, if they have not been already computed. By matching the current input symbol with one of the starter symbols the correct alternative can be selected, and the parsing proceeds in the normal way. If none of the starter symbols matches with the input, and there is an alternative that produces the empty string (assuming this can be decided), this alternative can be safely selected. If there is no such alternative, a syntax error must be reported. Obviously this works at least for LL(1) grammars: the fact that the parser makes a "default" move corresponding to the derivation of an empty string does not essentially change the behaviour of the parser. However, this scheme leads to an unnecessarily inefficient parser because the same current input symbol will be matched with a starter symbol many times on different nonterminal levels when the right-hand side of a production begins with another nonterminal. Note that when a starter symbol is matched with the current input in a nonterminal procedure, all the subsequent productions that are applied next to expand the leading nonterminal symbols on the right-hand sides of productions are in fact known in LL parsing. We call these productions the *left-corner productions* of the nonterminal in that context. So, our aim is a global data structure that supplies for each nonterminal not only pairs $(a, p)$ where $a$ is a starter symbol and $p$ is the production to be applied, but sequences of the form $(a, p_1, \ldots, p_k)$, where $p_1, \ldots, p_k$ is the sequence of the left-corner productions of the nonterminal in the parsing situation determined by the starter symbol $a$. The analysis procedures will then select the alternatives of the nonterminals according to this sequence, without consulting any more the current input symbol. The required data structure will be a labelled directed graph called the *start tree* of the nonterminal.

Suppose that the productions of each nonterminal are numbered $1, \ldots, n$; i.e. the alternative production rules of a nonterminal are given by unique numbers. The leaves of the start tree of $A$ will be the starter tokens of $A$, and some additional special symbols for handling empty derivations. The essential property of the start tree of $A$ is the following: if there is a leaf labelled $t$ (terminal symbol), then the labels of the arcs on the path from this leaf node to the root give the (numbers of the) left-corner productions when an $A$ produces something that begins with $t$. In additi-

on, if there is a leaf labelled $\langle A \rangle$, the labels of the arcs on the path from this leaf to the root give the (numbers of the) left-corner productions when an $A$ produces the empty string.

Assuming that we know how to build the start trees we can parse as follows. When the analyzer procedure of the nonterminal $A$ is called, we first check whether the start tree of $A$ is already constructed. If not, we construct it. Then the current input symbol is compared with the leaves of the start tree of $A$. If it matches with one of the leaves, the production numbers found on the path from the leaf to the root (in the reverse order) are applied in the subsequent activations of analyzer procedures of other nonterminals without consulting the current input symbol, until all these production numbers are consumed. If there is no match, but one of the leaves is $\langle A \rangle$, we know that $A$ produces the empty string and this is the only possible correct choice in this context. Hence we use the production numbers on the path from this leaf to the root as before. If there is no match and no $\langle A \rangle$ leaf, we report a syntax error. When all the production numbers have been consumed, the parser switches its "mode" and starts to process the right-hand side of the last selected rule in the normal way.

Here we will not discuss the construction of the start trees in detail (see [Kos 89]), but instead we show how to write the analyzer procedures. For that purpose we use some notations:

PrimaryStarters($A$) = $\{t$ in $V_T|$ there are productions $X_0 \rightarrow X_1 \ldots, \ldots, X_{k-1} \rightarrow X_k \ldots,$ where $k>0$, $X_0=A$, $X_k=t\}$;

Path($A$, $x$) = the sequence of numbers associated with the arcs from the (leaf) node $x$ to the root in the start tree of $A$, in the reverse order;

Variants($A$) = the number of alternative productions for the nonterminal $A$;

RhsLength($A$, $i$) = the number of terminal and nonterminal occurrences on the right-hand side of $A$'s production $i$;

RhsItem($A$, $i$, $j$) = the $j$th terminal or nonterminal occurrence on the right-hand side of $A$'s production $i$;

Sym($A$, $i$, $j$) = the terminal or nonterminal symbol corresponding to RhsItem($A$, $i$, $j$).

Further, we use the following procedures that are assumed to be provided by a general support module called MLI:

```
procedure Rule(): Integer;
var ProdNumber: Integer;
begin
    ProdNumber:=Head(LeftCorners);
    LeftCorners:=Tail(LeftCorners);
    if LeftCorners is empty then Mode:=Examine; end;
    return ProdNumber;
end Rule;
```

```
procedure Scan(t: Token);
begin
    if t is in the current input position then
        advance the input pointer past token t;
    else SyntaxError;
    end;
end Start;
```

For each nonterminal $A$ we construct procedure Create as follows (the choice of the name will become understandable later), to be included in the module of the nonterminal:

```
procedure Create;
begin
    if MLI.Mode=MLI.Examine then
        if there is no start tree for A then construct the start tree of A end;
        if there is terminal t such that
            a) t is in the current input position, and
            b) t is a leaf node of the start tree of A
        then
            if t belongs to Primary Starters(A) then
                advance the input position past t;
            end;
            MLI.LeftCorners:=Path(A, t);
        else
            if there is ⟨A⟩ in the leaves of the start tree of A then
                MLI.LeftCorners:=Path(A, ⟨A⟩)
            else
                MLI.SyntaxError;
            end;
        end;
        MLI.Mode:=MLI.Parsing;
    end;
    case MLI.Rule() of
        [for i in 1..Variants(A):
        i: [for j in 1..RhsLength(A, i):
            [if RhsItem(A, i, j) is terminal and j>1:
                [let S=Sym(A, i, j): MLI.Scan(S);]]
            [if RhsItem(A, i, j) is nonterminal:
                [let S=Sym(A, i, j): S.Create;]]]]
    end;
end Create;
```

Here LeftCorners and Mode are global variables provided by the general support module MLI, initially Mode = Examine. Note that we pay no attention to error recovery. Traditional error recovery techniques are in general not applicable, because there is no global grammar information that could be used e.g. to skip tokens after an error.

A modular recursive descent parser has some interesting properties. The fact that we make a default move leading to empty derivation implies that even some non-LL(1) grammars can be parsed succesfully. For example, the classical dangling else problem is solved simply by parsing according to the productions

IfStatement → "if" Expr "then" Statement ElsePart
ElsePart → "else" Statement |

which makes the grammar ambiguous. In this case the parser will always try to recognize a non-empty else part for the innermost preceding "if" instead of an empty one, if possible.

Another interesting feature is due to the fact that there is no global scanner, but the scanner is distributed in the start trees. This leads to "syntax-directed" scanning: only those tokens are considered in the scanner that are possible in the syntactic context. For example, consider the well-known Pascal subrange problem: the scanning of a subrange definition, say "1...10", fails because the principle of maximal length forces the scanner to expext a real constant after reading "1.". The problem does not appear in our method because a real constant cannot start a subrange and will not be considered at all.

Our method introduces also some new problems. Note that in principle the LL(1)-ness of the grammar is never known in advance, when the parsing begins (indeed, as shown above, the grammar need not be LL(1) in some cases.) Since only those parts of the grammar are examined that are actually used in analyzing a particular input, the LL(1)-ness cannot be decided at all. The method guarantees a correct parse for all LL(1) grammars, and the parser cannot accept an invalid input for any grammar, but it can a) produce a correct parse for some non-LL(1) grammars and b) report a syntactic error for a correct input of some non-LL(1) grammars. Problem b) is of course unpleasent: it would be more appropriate to report a grammar error than a syntactic error. Although most of the non-LL(1) cases must be eliminated during parsing in order to construct the start trees in a sensible way, some cases remain undetected. For a discussion, see [Kos89].

The syntax-directed scanning scheme implies certain problems, too. Because the scanner is distributed in the start trees, there may be conflicts between the tokens that are not known by the scanning process. For example, it is in general impossible to prevent a keyword belonging to one part of the grammar to be interpreted as an identifier when processing another part. Our method supports the convention that keywords are not reserved symbols but can be used e.g. as identifiers, as long as the left context determines uniquely the identity of the token.

The method described above has been implemented and some preliminary experiments have been carried out [Kos89]. The results show — somewhat surprisingly — that a modular scanner/parser is as fast as a traditional recursive descent one, reaching the speed of 300 000 tokens/min. It turns out that in practice the construction of start trees takes very little time: only for very small programs a difference in the running time was observed, when compared to a traditional recursive descent parser. The start trees tend to be rather small: for a subset of Pascal the average depth of the start trees was less than 2, and the average number of leaves was 2.7.

It is interesting to note that the behaviour of the modular parser is sensitive to

the properties of the grammar, and even to the properties of a particular input. If the grammar is "modular" in the sense that it consists of several relatively independent subgrammars, and only one or some of them are typically used in one input text, the start trees need to be constructed only for a small part of the grammar.

## 4. Construction of an abstract representation

We consider a program as a set of interrelated objects that, when put into a particular environment, behaves in a certain way implied by the language semantics. Consequently, there are two kinds of concepts involved in language implementation: *program concepts* that have more or less obvious concrete counterparts in the syntax of the language, and *environmental concepts* that are not represented in the program text, but belong to the "abstract machine" that executes the program. Our intention is to view both kinds of concepts in the object-oriented setting; correspondingly, instead of concepts we will speak of *program classes* and *environmental classes.* The program classes will be implemented by regarding nonterminals as classes, and by adding certain parts into the nonterminal modules constructed in Section 3. We shall use the term *nonterminal class* as a synonym for program class. The environmental classes could be provided by some general implementation support module (like MLI, see Section 3), or they could be implemented by additional modules; we use the latter approach in the sequel. The connection between these two class categories is established by the fact that some nonterminal classes are considered as subclasses of the environmental classes.

Let us first consider the problem of constructing an abstract representation of a program. To establish a sensible class hierarchy for the classes represented by nonterminal symbols we require that the syntactic specification is given in a certain form.

A context-free grammar is *structured*[1] if for each nonterminal $A$, either

(i) there is only one production that has $A$ on the left-hand side, or
(ii) all the productions that have $A$ on the left-hand side are chain productions;

but not both. Further, we say that a grammar is *well-structured,* if it has the following properties:

(i) it is structured;
(ii) it is reduced and non-circular;
(iii) there is no nonterminal $A$ such that the only production having $A$ on the left-hand side is a chain production;
(iv) each nonterminal appears on the right-hand side of a chain production at most once.

The basic idea is to interpret chain productions as presentations of class hierarchies. This is a natural interpretation: the fact that a nonterminal $A$ has the productions $A \to B_1, \ldots, A \to B_k$ is just another way of saying that a $B_1$ is an $A, \ldots,$

[1] This grammar form has been used (independently) by Jürgen Uhl [Uhl 86]. However, he used this form for establishing equivalence relations between nonterminals rather than class hierarchies. We adopt his term ("strukturierte Grammatik").

a $B_k$ is an $A$. A production that is not a chain production expresses only the constituent parts of a concept that is "basic" in the sense that it does not have subclasses, whereas a chain production $A \rightarrow B$ expresses the relation "$B$ is a subclass of $A$".

We say that the nonterminals having only chain productions are *superclass nonterminals*, and the nonterminals having no chain productions are *basic nonterminals*. The properties listed above for well-structuredness guarantee that

(a) each nonterminal is either a superclass nonterminal or a basic nonterminal, but not both;
(b) there are no needless or circular classes;
(c) there are no identical classes;
(d) the class structure is purely hierarchical (i.e. there is no multi-inheritance).

The properties (a), (b), (c) ,and (d) are implied by (i), (ii), (iii), and (iv), respectively. In the following we assume that a grammar is well-structured. Note that the well-structuredness of a grammar is easy to check using well-known techniques, and that an arbitrary context-free grammar can be automatically transformed into a well-structured one in a straightforward way, without affecting the essential properties of the grammar (like the generated language or the parsing properties); this requires only the introduction of some new nonterminals and possibly some renaming of the nonterminals. Also note that although class nonterminals cannot be circular they can be (and often are) recursive: there is no reason why a class nonterminal could not appear on the right-hand side of a production of one of its subclass basic nonterminals. The reader is invited to confirm that no nonterminal can appear both on the left-hand side and on right-hand side of some production (i.e. there are no directly recursive nonterminals).

Basically, the existance of an instance of a basic nonterminal in a syntax tree implies the existance of an object of the class corresponding to the nonterminal. In contrast, an instance of a superclass nonterminal merely establishes a new class level for an object that corresponds to the basic nonterminal instance somewhere below the superclass nonterminal.

To express classes in a program, we assume an Oberon-like [Wir88] type extension facility[2]: a record type may be extended with additional fields to create a new record type (subclass) that is upwards compatible with the original record type (superclass). Type extension is given as

**type** $T =$ **record** ($U$)... fields...**end**;

where $U$ is the superclass type that is extended with the new fields, yielding the subclass type $T$. As in Oberon, if a record type is given in the definition part of a module, it can be extended in the implementation part; this is only a means to introduce "invisible" fields for a visible record type. This minor feature turns out to be very useful in our method.

Consider the nonterminal modules constructed following the method described in the previous chapter. For each nonterminal module we specify a record type that provides all the local data for objects of the nonterminal class; we call this the *instance*

[2] The new object-oriented extension of Oberon [MTG 89] might have been even more suitable for our purposes; but we stick to a presentation language that is close to Oberon because we assume it is widely known.

*type* of the nonterminal. If the nonterminal has a superclass, this type is defined as an extension of the corresponding type in the module of the (*immediate*) superclass nonterminal. We will modify the "Create" procedure introduced in Section 2 so that it will return as its value a reference to the instance object. In the case of a superclass nonterminal the reference to be returned is provided directly by a call of a "Create" procedure of a subclass nonterminal; in the case of a basic nonterminal the instance object is explicitly created.

In an abstract representation of a program, certain fields of the instance type refer to objects whose (instance) type is provided by the modules of the constituent nonterminals. These fields can be declared only in the body of the module: the fact that nonterminals may be recursive prohibits the declaration of these fields in the definition part (otherwise there would be circular importing between the definition parts). This is natural also because these fields are internal knowledge of the objects that should be used only by the methods of the corresponding class (i.e. by the procedures of the module). However, the instance type itself must be declared in the definition part of the module, because it is needed by other modules. This contradiction can be nicely solved using the Oberon-like feature which allows the adding of new "invisible" fields in the module body into a record given in the definition part.

In addition to the special notations introduced in Section 3, we use the following notation:

Super($A$) denotes the immediate superclass nonterminal of $A$, if it exists; otherwise $A$;

In the following we give a scheme for generating a nonterminal module together with the parts that are needed for constructing an abstract representation of the program.

```
definition NontName;
    import MLI
        [if NontName has a superclass: , [let S=Super(NontName): S]];
    type Class=pointer to InstanceType;
    type Instance Type=record
        [if NontName has a superclass:
            ([let S=Super(NontName): S.InstanceType])]
    end;

    var Descriptor: MLI.DescriptorType;
    procedure Prepare;
    procedure Create(): Class;
end NontName;

module NontName;
    import MLI
    [for N in {A | there is a production NontName→ ... A ...}: ,N];
    [if NontName is a basic nonterminal:
    type InstanceType=record
        [for j in 1..RhsLength(NontName, I):
            [if RhsItem(NontName, I, j) is nonterminal:
```

```
            [let Y=Sym(NontName, l, j), Z=j:
            comp_Z_Y: Y.Class;]]]
    end;]

    procedure Prepare;
        ... for constructing the start tree, see Section 3 ...
    end Prepare;
    procedure Create(): Class;
        var NewObj: Class;
    begin
        [if NontName is a superclass nonterminal:
        if MLI.Mode = MLI.Examine then
            ... as in Section 3 ...
        end;

        case MLI.Rule() of
            [for i in 1..Variants(NontName):
                [let X = Sym(NontName,i, l):
                    i: return X.Create();]]
        end;]

        [if NontName is a basic nonterminal:
            New(NewObj);
            [for j in 1..RhsLength(NontName,l):
                [if RhsItem(NontName,l,j) is terminal:
                    [let t = Sym(NontName,l,j): MLI.Scan(t);]]
                [if RhsItem(NontName,l,j) is nonterminal:
                    [let Y = Sym(NontName,l,j), Z = j:
                        NewObj^.comp_Z_Y:= Y.Create();]]]
            return NewObj;]
    end Create;
end NontName;
```

Note that we have slightly modified the parsing scheme presented in Section 3 to make use of the well-structuredness of the grammar. Since a basic nonterminal has only one syntactic alternative, there is no need for a **case** statement and for the preceding **if** statement in the Create procedure. Hence these statements can be omitted, provided that the arcs corresponding to the productions of basic nonterminals are removed from the start trees as well.

The above scheme produces a structure which is exactly the abstract syntax tree of the program. However, we are aiming at a more elaborated structure that would be more amenable to further processing. For this purpose we need new environmental classes.

As an example, suppose that we have an environmental class providing the abstract concept of a general list. To be able to conveniently specify the sequential execution of a statement list we would like to represent a statement list as a list rather than as a tree structure. Hence, we say that the nonterminal class "StatementList" is a subclass of the environmental list class. Consequently, the nonterminal class that gives the element of the list ("Statement") must be a subclass of another

environmental class that gives the abstract concept of a list element. Since these environmental classes are obviously closely related, they are provided by the same module:

```
definition List;
    type List=record end;      (* only invisible fields *)
    type Elem=record end;
    type ListClass=pointer to List;
    type ElemClass=pointer to Elem;
    procedure CreateList(): ListClass;
    procedure Insert(L: ListClass; E: ElemClass);
    ... other procedures ...
end List;
```

We make use of these classes in the instance types of StatementList,

```
type InstanceType=record (List.List) end;
```

and Statement,

```
type InstanceType=record (List.Elem) end;
```

Note that in principle the environmental superclasses are treated in the same way as nonterminal superclasses. However, in StatementList there are no (invisible) fields of the instance type that would contribute to the abstract representation; the structure of a statement list is implicitly accessible through the operations provided by the list module. An element of a list (Statement) is created normally using New in the creation operation of the basic nonterminal (e.g. IfStatement), and then inserted into the list using the appropriate list operation. In contrast, a list (StatementList) must be created using directly the creation operation provided by module List because this requires certain initializing actions that cannot be given in the nonterminal module.

Note that the class hierarchy must be consistent in the sense that all the instances of a nonterminal class $X$ have the same class levels, independently of the context. The class levels of the objects do not depend on the syntactic context, but only on the existance of certain chain productions. Hence, even though $X$ is not produced by its superclass nonterminal $Y$ in a particular context, the object created for the instance of $X$ has a $Y$-level. This holds for environmental superclasses as well: for example, a statement has to be a list element in every context, even though it is (syntactically) not an element of a statement list.

Since we regard a list element as a superclass of a statement, this must be true for every instance of a statement: the class hierarchy must be consistent in this sense. Hence a statement should always appear in a list of statements.

Let us consider a more complicated example, the implementation of name environments (i.e. symbol tables). Again we may assume the existance of an additional module providing certain environmental superclasses. For example, we could have:

```
definition NameEnv;
    import ... ;
    type Decl=record name: String; end;
    type Region=record ... end;
```

```
    type DeclClass=pointer to Decl;
    type RegionClass=pointer to Region;
    procedure CreateRegion(): RegionClass;
    procedure DeleteRegion(X: RegionClass);
...
end NameEnv;
```

Suppose that we have the grammar fragment

```
Declaration = VariableDeclaration|TypeDeclaration|...
VariableDeclaration = "var"...
TypeDeclaration = "type"...
```

Nonterminal Declaration (or its instance type) should then be a subclass of Decl, and the nonterminals generating visibility regions like modules or blocks should be subclasses of Region; VariableDeclaration and TypeDeclaration are subclasses of Declaration as usual. The creation operation of VariableDeclaration (a basic nonterminal) creates a new object in the normal way, and then inserts it to the region using an appropriate operation provided by NameEnv. The creation operation of a region nonterminal (say, Block) also creates the region object using New, but it must also apply other operations provided by NameEnv to "enter" and "exit" the region.

It should be noted that above we have only sketched the basic guidelines that could be followed in the implementation. The details depend on the source language, and it is possible that even the basic principles may have to be adjusted to fit a particular language.

## 5. Semantics

The (dynamic) semantics of a language is essentially more irregular than the parts discussed previously. Hence it is difficult to develop techniques that would be generally applicable. The basic principle, however, should be that the dynamic semantics of the instances of nonterminal classes should be based on the methods of the classes. We illustrate this by an example.

Consider the following fragment of a language:

```
Statement→AssStatement|IfStatement|...
AssStatement→VariableDenotation :=" Expression
IfStatement→"if" Expression "then" Statement
```

Here Statement is a superclass nonterminal, while AssStatement and IfStatement are basic nonterminals. Each statement has the property that it can be executed; hence a semantic field of the instance type of Statement provides a procedure (method) for executing a statement object.

```
definition Statement;
    import MLI;
    type Class=pointer of InstanceType;
```

```
    type StatEx=procedure(X: Class);
    type InstanceType=record
        execute: StatEx;
    end;
    ...
end Statement;
```

Module Statement does not provide any value for field "execute"; using the object-oriented terminology this is a virtual method of the class Statement. The value of "execute" is given at the lower level where the kind of the statement is known:

```
definition IfStatement;
    import MLI, Statement;
    type InstanceType=record (Statement.InstanceType) end;
    type Class=pointer to InstanceType;
    procedure Create(): Class;
    ...
end IfStatement;
module IfStatement;
    import MLI, Statement, Expression;
    type InstanceType=record
        comp_1_Expression: Expression.Class;
        comp_2_Statement: Statement.Class;
    end;

    procedure ExecuteIf(S: Statement.Class);
    begin
        with S: Class do
            if S^.comp_1_Expression^.evaluate()=1 (* true *)
            then S^.comp_2_Statement^.execute
        end
    end ExecuteIf;
    procedure Create(): Class;
        var NewObj: Class;
    begin
        New(NewObj);
        NewObj^.execute:=ExecuteIf; (* determine the execution method *)
        Scan("if");
        NewObj^.comp_1_Expression:=Expression.Create();
        Scan("then");
        NewObj^.comp_2_Statement:=Statement.Create();
        return NewObj;
    end Create;
    ...
end IfStatement;
```

In this way every creation of a statement instance, carried out by the basic statement nonterminals like IfStatement, must assign an appropriate value for the

execution operation. Hence, when the execute-field of a statement object is called, the actual routine will depend on the kind of the statement. We have followed here the Oberon conventions which require that the actual procedure has the same parameter types as the virtual one; therefore IfStatement's parameter has to be of type Statement. Class (and not Class which would be more natural). Explicit subclass checking (**with** statement) guarantees that the parameter statement is really an **if** statement.

## 6. Discussion

The starting point of this work has been the observation that so far the modularization of language implementation software has been based on very implementation-oriented thinking. Implementation aspects have always had deep effect on the way we design and view programming languages. We argue that the conventional modularization technique which treats the source language as a black box has led to the view that languages are in principle indivisable, and that it is not sensible to try to reuse parts of existing language implementation software in the development of other languages. It is characteristic that programming languages are often regarded as a means to communicate with a computer, as a "formal language", suggesting a close relationship with natural languages. However, programming languages are not like natural languages: they are most of all technical tools to build systems. Like other complex industrial tools they should be composed of relatively specialized parts that can nevertheless be used as such in many kinds of system building tools. This would give us the same benefits that are now regarded as self-evident in other engineering branches: new production (i.e. programming) systems could be rapidly developed for different purposes using existing building blocks, old systems could be modernized by replacing certain parts with more advanced parts, and system maintenance would be easy because the system consists of small modules with clean interfaces.

## References

[Gro 84] GROSSMANN R., HUTSCHENREITER J., LAMPE J., LÖTZSCH J., MAGER K.: Depot 2a — Metasystem für die Analyse und Verarbeitung Verbundener Fachsprachen. Anwenderhandbuch, Sektion Mathematik, Technische Universität Dresden, 1984.

[HKR 88] HEERING J., KLINT P., REKERS J. G.: Incremental Generation of Parsers. Report CS—R8822, Centre for Mathematics and Computer Science (CWI), Amsterdam 1988 (also in the Proc. of Sigplan '89 Symposium on Design and Implementation of Programming Languages).

[HeR 75] HEINDEL L. E., ROBERTO J. T.: Lang-Pak — An Interactive Language Design System. Elsevier, New York—London—Amsterdam 1975.

[Kos 88] KOSKIMIES K.: Software Engineering Aspects in Language Implementation. In: Proc. of Workshop on Compiler-Compiler and High-Speed Compilation, Berlin, Oct. 1988.

[Kos 89] KOSKIMIES K.: Lazy Recursive Descent Parsing for Modular Language Implementation. Arbeitspapiere der GMD 376, Gesellschaft für Mathematik und Datenverarbeitung, Forschungsstelle Karlsruhe, April 1989.

[MTG 89] MÖSSENBÖCK H., TEMPL J., GRIESEMER R.: Object Oberon — An Object-Oriented Extension of Oberon. ETH Zürich, Institut für Computersysteme, Report 109 (June 1989).
[Toc 88] TOCZKI J., GYIMOTHY T., HORVATH T., KOCSIS F.: Generating Modular Compilers in PROF-LP. In: Proc. of Workshop on Compiler-Compiler and High-Speed Compilation, Berlin, Oct. 1988.
[Uhl 86] UHL J.: Spezifikation von Programmiersprachen und Übersetzern. GMD-Bericht Nr. 161, Gesellschaft für Matematik und Datenverarbeitung, 1986.
[Wat 85] WATT D. A.: Modular Description of Programming Languages. Report A-81-734, Computer Science Division — EECS, University of California, Berkeley 1985.
(WeM 80] WELSH J., MCKEAG M.: Structured System Programming. Prentice-Hall 1980.
(Wir 88] WIRTH N.: The Programming Language Oberon. Software Practice and Experience 18 (7). 671—690 (July 1988).