COMPUTER-AIDED LANGUAGE IMPLEMENTATION WITH TaLE

Esa Järnvall and Kai Koskimies

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1993-4

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1993-4, JULY 1993

COMPUTER-AIDED LANGUAGE IMPLEMENTATION WITH TaLE

Esa Järnvall and Kai Koskimies

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

ISBN 951-44-3398-X
ISSN 0783-6910

Computer-Aided Language Implementation with TaLE

Esa Järnvall and Kai Koskimies
Department of Computer Science, University of Tampere,
P.O. Box 607, SF 33101 Tampere, Finland

email: ejj@cs.uta.fi, koskimie@cs.uta.fi

Abstract

TaLE is a specialized editor for developing language implementations in an object-oriented programming environment. In contrast to conventional language implementation systems, there is no formal metalanguage for specifying a language; instead, the user edits the classes taking part in the implementation under the control of a specialized editor. This editor provides a high-level, partly graphical view of the classes representing language structures. The system supports the reuse and refinement of the language implementation classes, incremental implementation development, integration of syntactic and name analysis, and special views for classes representing standard language features. The expected main advantages of the metalanguageless approach and the graphical user interface are high usability and fast development cycle.

1 Introduction

TaLE (Tampere Language Editor) is a new tool supporting the development of language implementation software in an object-oriented programming environment. The design of TaLE is unconventional in the sense that TaLE emphasizes software engineering qualities rather than contributions in formal language specification; this makes the system in many ways different from more traditional language implementation systems. In fact, TaLE employs a different paradigm: the user is not expected to write a language specification, but to edit the classes representing language structures under the control of a specialized editor. Currently the target language (that is, the language of the produced software) is Eiffel [Mey88], but in principle this language could be any class-based object-oriented language. The system has been implemented in Eiffel 2.3 and is currently running in Sun3/UNIX.

TaLE is particularly intended for the rapid implementation of application-oriented languages, i.e. for various nontrivial textual representations of data, specifications, algorithms etc.; typically these are "little languages" as proposed by Bentley [Ben86]. Since such a language is often only a small aspect in a large software project, we cannot expect that the user of a language implementation system would be willing to learn a new formal metalanguage based on a theoretically oriented specification paradigm. The simpler and the more self-explaining the user interface of a language implementation system is, the smaller tasks will be included in the potential applications of the system. Our intention is that an average programmer will decide to use our system for virtually all nontrivial processing of structured textual data, in the same way as GUI editors are currently used for implementing graphical user interfaces. This puts high demands on the user-friendliness and simplicity of the system. In TaLE, these requirements are hoped to be satisfied through the metalanguageless, editor-based approach, and through high-level, intuitive views of the classes taking part in the implementation.

Conventional language implementation systems are typically closed: they provide a mapping from an abstract specification into an executable language processor written in a target language, and the user is not expected to understand the target language, far less modify the resulting processor. However, this idealistic view is rarely fully possible in practice because the abstract metalanguage is not general or efficient enough, or because the produced language implementation is part of a larger system, and the resulting code must be patched to integrate it with the rest of the software. The editor-based approach of TaLE leads to an open system in which the use of the base language is natural and in which the user essentially gets what s/he sees. A necessary requirement for this approach is the object-oriented programming paradigm which allows a close relationship with the concepts of the implemented language and the software units (classes) of the implementation.

In the long run, the crucial factor of the productivity in software development is reuse. In language implementation this aspect has been mostly ignored (at least in the usual sense of software reuse), because programming languages have been regarded as indivisible entities that give little opportunities for sharing common code. However, actually this is not true: there is a lot of almost identical features in different languages, and it seems reasonable to assume that this could be reflected in the implementation as reusable units. For example, practically every language has a notion of an arithmetic expression, with minor variations. This implies that essentially the same concept is implemented over and over again, and that very similar code is repeated in numerous language processors. The same holds for concepts like standard constant denotations, control structures, subprogram mechanisms, type systems etc. The possibility to reuse code is particularly obvious for special-purpose languages that are under design: in many cases it would be sufficient to simply pick up a suitable standard form of, say, arithmetic expressions from a library of standard language features, in the same way one employs a standard data structure from a general library. Even if a direct adoption of a library feature is not appropriate, it should be possible to easily modify and extend a library unit according to the needs of a particular language.

In fact, certain kinds of reuse are fairly common, although perhaps not identified as such: special-purpose languages are often developed by extending a general-purpose language with application-specific features, or by embedding a particular structure from a general-purpose base language within a special-purpose language. In both cases one actually reuses all or some of the structures of a general-purpose language, in the hope that they need not be re implemented. Here we want to generalize this kind of language development, and regard languages as collections of relatively independent, replaceable units.

TaLE supports the reuse of language concepts and structures in three ways: first, by employing a distributed implementation model in which language structures are implemented by highly independent classes TaLE allows a language to adopt structures (and their implementations) from other languages; second, general language-independent concepts can be implemented on an abstract level and refined for individual languages (making use of the subclassing mechanism); third, standard language concepts and their implementations are built into the system, so that a user can directly exploit them in his/her language, tailored through high-level specification interfaces. Together with the high-level views provided by the metalanguageless user interface, the facilities supporting reuse are expected to speed up the language development process in most cases by an order of magnitude when compared to traditional systems like LEX/YACC.

This paper is an overview of the design principles of TaLE. In the following section we briefly discuss the related research. In Section 3 we present a simple application of the system, demonstrating some of its key features. The underlying basic model of formal languages is discussed in Section 4. The implementation and efficiency issues are discussed in Section 5. We conclude with some remarks concerning future work.

2 Related work

Although the overall character of TaLE is unique (as far as we know), there are similarities with existing systems. Many authors have applied object-orientation in the specification of formal languages (or programming environments) during the last few years. Object-oriented forms of a context-free grammar have been introduced more or less independently by Tenma et al [Ten88] ("AND/OR rules"), Lehrmann Madsen and Nørgaard [LMN88] ("structured CFG"), and Koskimies [Kos88] ("well-structured CFG"), and later by other authors (e.g. [WuW92]: "restricted CFG"); these forms are roughly equivalent ways of interpreting syntactic alternation as subclassing. Our language model discussed in Section 4 contains this aspect, too. Attribute grammar based object-oriented language specification methods have been used by Hedin [Hed89], Grosch [Gro90], and Shinoda and Katayama [ShK90]. A (weakly) object-oriented language implementation system is presented by Koskimies and Paakki [KoP87], representing a relatively loose integration of object-oriented programming and one-pass attribute grammars. Since TaLE is not based on attribute grammars and has no specification language, the similarities with these systems are limited to the object-oriented view of language processing. Such a view is also presented in the Muir language development environment [Win87]; differences in the language model are mainly due to our emphasis on reusable language structures. Another major difference is again that we abandon a formal metalanguage, and that we concentrate on the implementation of a language itself, rather than on the implementation of its environment. A completely different model of object-oriented grammars is given in [AMH90], where a grammar as a whole (rather than individual language structures) is subject to inheritance. Object-oriented language implementation techniques are discussed also in [Gra92] both for a particular language (Smalltalk) and for a generator system; this work deviates from our implementation model in that the design is structured according to language-independent concepts whereas we apply a language-oriented structuring. The reason for our decision is incrementality, to be discussed in Section 5.

Due to the interactive nature of TaLE, it is essential that language implementations can be constructed incrementally, that is, changing one part in a language should not necessitate the reconstruction of the whole implementation. In particular, this requirement is in conflict with conventional parser generator techniques which assume that the whole grammar is given as input for the generator. Incremental (non-object-oriented) parsing techniques have been studied recently by several authors, for varying reasons. A lazy, incremental recursive descent parsing method is presented in [Kos90]; this method has served as a starting point in the design of parsing in TaLE. The main difference with other incremental (deterministic) parser generator techniques ([GHK88], [Hor89] and [HKR89]) is that we use program-formed (recursive descent style) parsers rather than table-driven ones.

Many details in the user interface of TaLE have been inspired by existing systems. The "editing-without-typing" paradigm of the Orm programming environment [Mag90] has been followed to some extent in TaLE: the typing of an identifier can always be avoided when the identifier is already known; the user can simply select the identifier from a context-sensitive dynamic menu. The class browser of TaLE has been designed using the Smalltalk programming environment [Gol84] as a starting point.

3 An introductory example

We will first give a general impression of TaLE using a small example. Although the example language is by no means a realistic one or a typical TaLE application, it is sufficiently interesting for demonstrating purposes. Our example language is a simple desk calculator language: we want to be able to express sequences consisting of assignments to named variables and output instructions, e.g.:

X:= 55; Y:= (X+24)*X; Z:= X*Y; OUT Z+220

The produced language processor should carry out the computation of the expressions and print the expression in the output instruction. We assume that a variable can be referenced only if its value has been defined before; hence each assignment and output instruction can be "executed" immediately after it has been analyzed.

Individual language structures are implemented by classes in TaLE; in conventional terminology, a class corresponds to a nonterminal (this is only a rough correspondence, we will return to this later). The first task of the language implementer is to consider the existing classes provided by TaLE either as built-in classes or as classes of some other language: is there something we could directly apply in our language? The TaLE class browser (Fig. 1) shows e.g. classes Std_Expression (standard arithmetic expressions) and Std_List (a standard list structure) which seem to be useful to us. Further, Pascal_id and Pascal_int (subclasses of Identifier and Std_integer) could be candidates for expressing identifiers and integer constants, and Simple_output seems a promising class for implementing the output instruction. We can hope to be able to use these classes either directly or as superclasses in our language.

We use the following textual notations in this paper: if a class (say A) is defined as a subclass of another class (say B), we write

A = B(...)

where the parenthesized part contains the additional specifications of the subclass. The form of the specifications depend on the class in question; here we will give them as an informal list of feature specifications. If A has subclasses B_1,...,B_k, we write

A > B_1 | ... | B_k

If the objects of class A has components of class B_1, ... , B_k (in this order), we write

A -> B_1 ... B_k

Our "plan" to implement the example language could be now presented as follows:

Program = Std_list(element: Instruction; separator: ";")
Instruction > Variable | Output
Variable -> Pascal_id ":=" My_expression
Output = Simple_output(keyword: "OUT", body: My_expression)
My_expression = Std_expression(operators: +,-,*,/; types: integer)

Built-in high-level concepts

Let us first define a class for the expression structure we need in our example language. For this purpose we use the standard facilities provided by the predefined class Std_expression: we give a "create subclass" menu command for class Std_expression, and the special view for expressions is displayed to allow the user to construct the subclass with the given name (here My_expression). This is an example where a standard language notion is reused through a special interface allowing the fine-tuning of the concept.

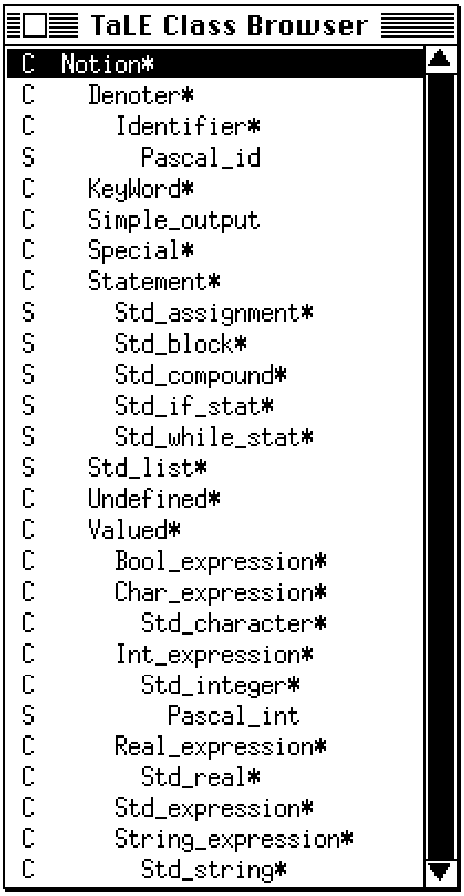

Fig.1. The class browser. "C" denotes "conceptual" and "S" denotes "structural" (see Section 4).

The expression view is shown in Fig. 2 after completion: the user has selected the operator symbols s/he wants, their precedence and associativeness, the allowed type combinations (shown in a separate window), the representations for constants (as classes; here we need only integer constants given as class Pascal_int), the name of the class giving the other primitive constituents of expressions (here we give a new, so far undefined class Variable_access for this purpose), and the parenthesis convention. Further, the user can specify whether s/he wants static type checking and/or static evaluation of the expression; in our simple example language we can decide to use static evaluation. If only standard operations +, -, *, /, are needed, this is all that has to be done; in the case of non-standard operators the user must give the Eiffel-statements that implement the operator in a separate text window.

Note that we applied here also another type of reuse: we adopted Pascal's integer representation (Pascal_int) directly as such in our language, with its full implementation. In this case the reused structure is a simple token, but in principle all structures (classes) are reusable in this way: due to our incremental approach a structure is not tied to a particular language but a more or less independent unit.

The view of the subclasses of Std_expression shown in Fig. 2 is an example of a *special view*, tailored for the particular properties of the class. Currently there are other special views for instance for the standard lexical structures (Identifier, Std_character, Std_integer, Std_real, Std_string) allowing certain structural properties to be individually selected. In principle special views can be designed for all sufficiently standard language features.

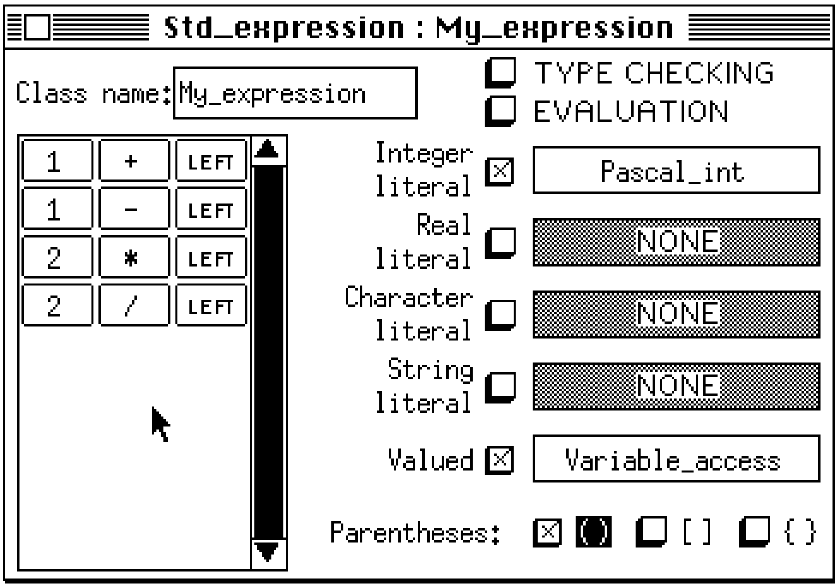

Fig. 2. The view for expression class My_expression.

Graphical class views

Let us next construct a class for the variable assignment structure (Variable -> Pascal_id ":=" My_expression). This class we will construct from scratch, rather than building on an existing reusable class. We ask the system to create a new structural subclass for the root class Notion; the system then displays a general structural class view for constructing the new class. This view is shown in Fig. 3 after some editing actions.

The upper section of the view contains the feature list, the check list, and buttons for giving the class certain special properties. The feature list contains all the (visible) attributes and operations of the class, including the inherited ones (the latter are associated with the defining class in brackets). The user may introduce new features; the Eiffel code of the features is given in a separate text window appearing when a feature is added or edited. In this class we need no user-given features.

The check list contains the semantic checks carried out during analysis (e.g. type checking). Each check is denoted in the list by a string which serves as the message emitted when this check fails; the check itself is given as a Boolean expression over the attributes of the class and the attributes of the components in the pattern. Here we need no checks, either.

The property buttons are actually short-cuts for inheriting certain predefined classes, but they also cause some additional actions to be carried out automatically by the system. Button "SCOPE" makes the class a static visibility region, i.e. the language structure it represents will be associated with a set of named objects. Button "NAMED" associates the class with the properties of a named object that can be stored in a built-in object base; each object with this property is automatically inserted into the object base as a member of the set associated with the smallest enclosing "SCOPE" structure. Button "VALUED" associates the class with a special value attribute of a predefined class; currently this class is capable of representing all the scalar values of Eiffel. Finally, button "TOKEN" turns the structure into a syntactic token in the sense that no spaces are allowed between the different parts of the structure, and that the parser uses this structure as a whole for syntactic look-ahead. This is the way arbitrary user-defined token categories can be introduced in TaLE, in addition to the (fairly covering) standard ones provided by predefined classes and their special views.

Since each variable assignment can be seen as a declaration of the left-hand side variable in our example language, we tick the NAMED button; consequently, the class inherits a string-valued attribute *key* containing the identifying name of the object. Likewise, we tick the VALUED button since we want that each assignment is associated with a value; as a result, the general *value* attribute is inherited from a system class.

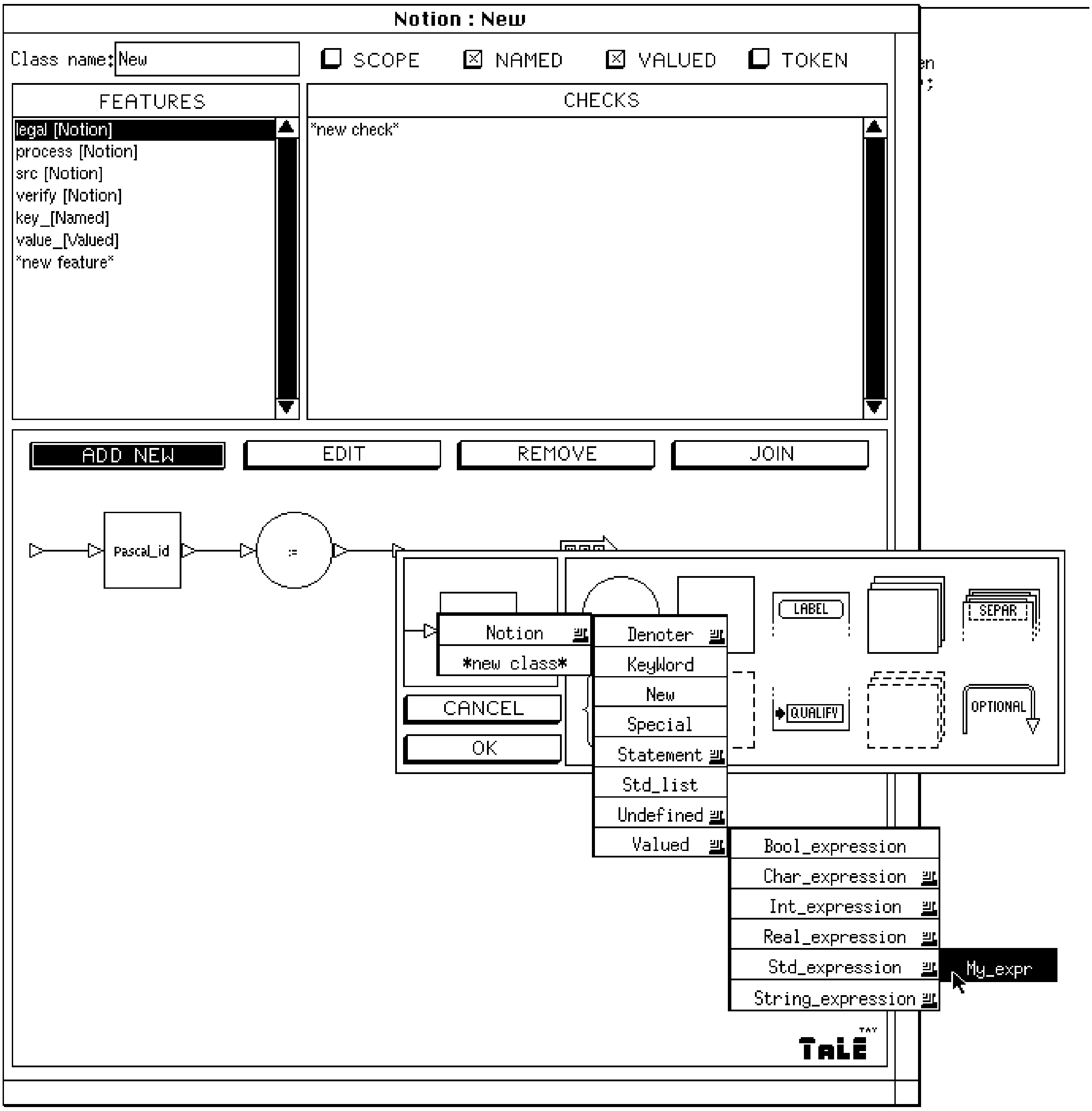

Fig. 3. The editing window for the assignment structure.

In the lower section of the view the syntactic structure is given graphically as a "railyard" syntax; we call this the *pattern* of the class. A class that has a pattern is called a *structural class*, other classes are called *conceptual classes*. The pattern is constructed and edited directly using mouse-driven commands and the mode buttons appearing in the upper part of the pattern section. The icons denoting components of a pattern are selected from a palette appearing when a component is added or edited. The view of Fig. 3 is shown in a situation where the user is constructing the pattern: the icons representing the left-hand side and the assignment symbol have been inserted, and the user has indicated s/he wants a new component (for the right-hand side expression) in the pattern; as a result, the palette is shown allowing the user to select the kind of the component and the class it represents. The class can be selected from a hierarchical, dynamic menu showing all the existing classes; here the user has selected class My_expression. The icons in the palette represent a single keyword, a single substructure, a list structure, a list separator, a set of alternative keywords, a secondary structure (a named substructure without a class of its own), a secondary list, and a passing arc (for making a component an optional one).

Each arrow head in the pattern represents a code location: the user may insert arbitrary Eiffel code into the pattern by clicking on an arrow head. This results in the opening of an Eiffel window in which the user may type any sequence of Eiffel statements. These statements will be executed during the analysis phase at the corresponding point. The system assists the user in the writing of the statements: the features of the component structures need not be explicitly written but they can be selected from a menu displayed when the corresponding pattern icon is pointed by the mouse. In this case we must define the value and key of a variable: we click on the last arrow head (since we want that these actions are carried out after the analysis of the whole assignment), and write the following text in the opened text window:

```
key := Pascal_id_s.src;
value := My_expression_s.value;
```

That is, the key attribute gets its value from the source string (src) corresponding to the Pascal_id component, and the value is taken directly from the value of the My_expression component. The system supports the writing of this code: references to the attributes of the component structures (e.g. "Pascal_id_s.src") can be produced by selecting the appropriate attribute from a dynamic menu appearing when the corresponding component icon is clicked with the mouse. Similarly, references to the attributes of the structure itself (e.g. "key") can be produced by selecting them from the feature list with the mouse. Finally, we give this class the name "Variable", and exit the class window.

Reusing an abstract general-purpose class

Let us next create a class for output statements, i.e. class Output. For output structures TaLE offers no built-in facilities, but what a lucky coincidence, somebody has previously constructed an abstract, general-purpose class Simple_output which we can now reuse. This is a conceptual class. Part of the existing specification of Simple_output is shown in Fig. 4.

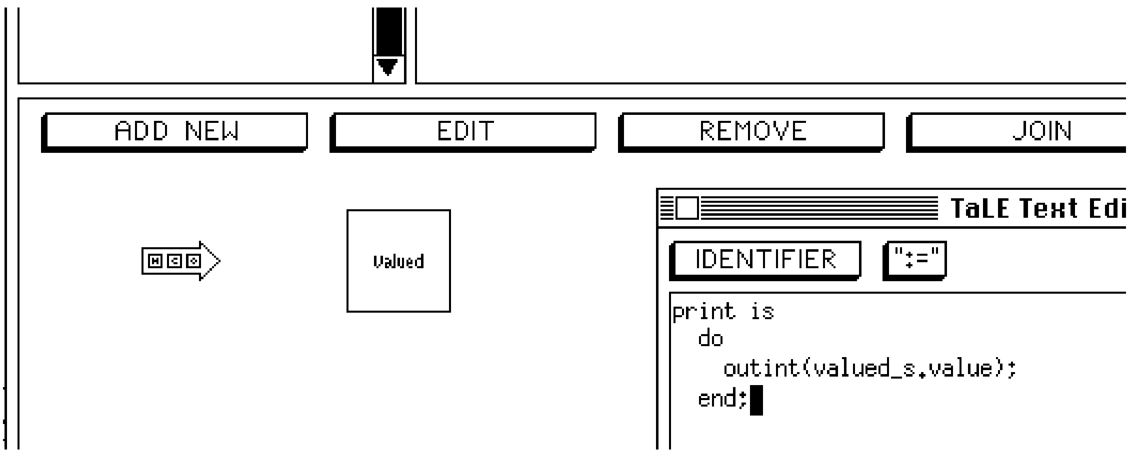

Fig. 4. Part of the class window for Simple_output.

Since Simple_output is a conceptual class, it does not have a pattern. Nevertheless, even a conceptual class can have components which exist independently of a pattern; we call them abstract components. In this case there is a single abstract component of class Valued representing the structure whose value is to be printed. In addition, the class defines an operation called print; this operation simply prints out the value attribute of the abstract component. The code for the print operation is shown in a separate text window.

We can now construct the class Output as a structural subclass of Simple_output. The view shown for this subclass contains initially the inherited abstract components, located in sequence after the thick arrow symbol. The user must "consume" all the inherited abstract components in the pattern of the new subclass, by dragging them into their place in the pattern. In this way the user associates each abstract component with some concrete component position in the pattern. When an abstract component has been inserted into a pattern, it can be further refined, i.e. its class can be narrowed from the original one. Fig. 5 is shown in a situation where the user has already inserted the keyword icon into the pattern (for "OUT"), and is next going to drag the abstract Valued-component into its place. This component must be further refined to class My_expression (which is a descendant class for Valued). Note that the inherited components are displayed with thick border, indicating that they cannot be edited in this view.

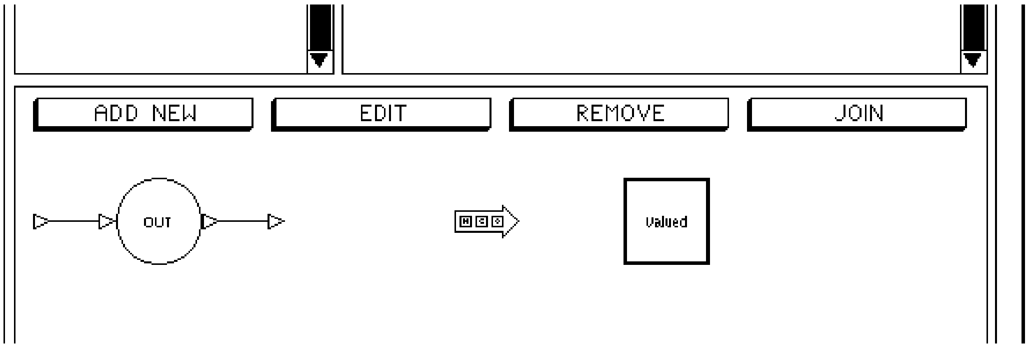

Fig. 5. Constructing the Output class.

The actual effect of an output instruction can be easily realized using the inherited print-operation: all we have to do is to insert the call of print into the last arrow head of the pattern of Output.

TaLE is an incremental system: each constructed class is a full-fledged Eiffel class after its editing has been completed. Usually a class need not even be recompiled when some other classes are modified, although the class makes use of the modified classes; this is due to the distributed implementation strategy of TaLE, to be discussed in Section 5. Since for each undefined class is given a default implementation (an undefined class X is assumed to have a pattern with a single keyword item "X"), it is possible to test a class at any point, even if it makes use of undefined classes.

We can test class Output by activating a special Test-command in the main menu. The opened tester window asks for the class to be tested. We select the class Output and write in the input pane:

OUT 2+3*4

We click on an activation button, and observe the result "14" in the output pane.

At this point we could construct class Instruction as well (recall Instruction > Variable | Output). Since the only purpose of this class in our language is to collect classes Variable and Output under a common name, it needs no features itself: everything will be specified in its subclasses. Therefore we create it as a (conceptual) subclass of the root class, Notion. Immediately after giving its name we exit the Instruction window, and return to the class browser level. There we use a special multiple inheritance command, forcing Variable and Output to inherit Instruction.

Automated name analysis

Let us next concentrate on the so far undefined class Variable_access, representing the variable references in an expression. This is a structural class: it has a particular syntactic form consisting of an identifier. We define it as a subclass of Notion and tick the VALUED button in the appearing view. We insert a single component to the pattern, and select the class Pascal_id for the component

(Fig. 6). Since this component must be associated with an existing variable, we *qualify* it with class Variable. In this way we make sure that the identifier indeed is the name of an existing Variable object. Using qualification, the association of named entities and their references in the source is carried out automatically by the system. An additional advantage of using qualification is that the parsing process can make use of the qualification information, and avoid LL(1) look-ahead conflicts that would otherwise arise (although in this case there is no fear of that).

We must further specify that the value of a Variable_access object is obtained from the value of the Variable object denoted by this object. We click on the arrow head at the end of the rail-yard syntax and write the following text in the opened text window:

```
v ?= denoted;
value:= v.value
```

where `denoted` is a predefined attribute that automatically refers to the Named (Variable) object associated with this object. Attribute `v` is a new feature that must be given to class Variable_access due to the type rules of Eiffel: since the static class of `denoted` is Named, it is not guaranteed that the denoted object would inherit Valued; therefore we must introduce an additional attribute `v` of type Valued, and apply so-called reverse assignment attempt ("?=") which checks the dynamic class of the right-hand side object (in this case the check never fails).

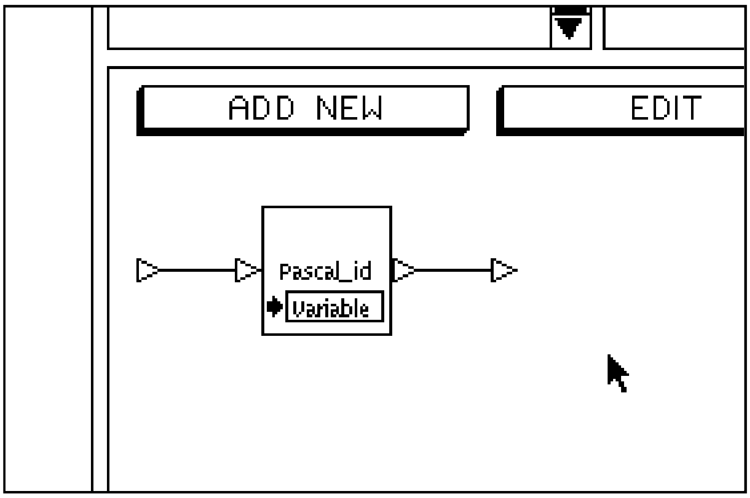

Fig. 6. The pattern for Variable_access, with a qualified component.

Refining an existing class

Since the entire "program" in our example language is a list of something, we define it as a subclass (named Program) of Std_list. When the "create subclass" command is issued for Std_list, the view in Fig. 7 is shown. This view is a so-called refinement view in which the original, inherited syntactic pattern is shown with thick border lines, indicating that it is not modifiable. What the user can do is to refine the components in the pattern, i.e. to narrow their classes. This is done by specifying the class in each refinement icon located under the actual component icon. In the inherited pattern of Std_list, the list element is specified to have class Notion, implying that any class will do here. The concept of Std_list also includes a separator, which is specified to be of class Keyword; this is a special built-in class whose subclasses are implicitly all individual key strings or sets of such strings (a key string is a class only in a technical sense, allowing conceptually unified handling of keywords). It is also possible to give new arrow head actions, features and checks in the refinement view.

In our example language the list element class is Instruction, and the separator symbol is semicolon. Fig. 7 is shown in a situation where the user has already selected Instruction as the refinement class of the Notion component, and is currently refining the separator component. Here these refinements are sufficient for the complete specification of class Program, and this concludes the implementation of our example language.

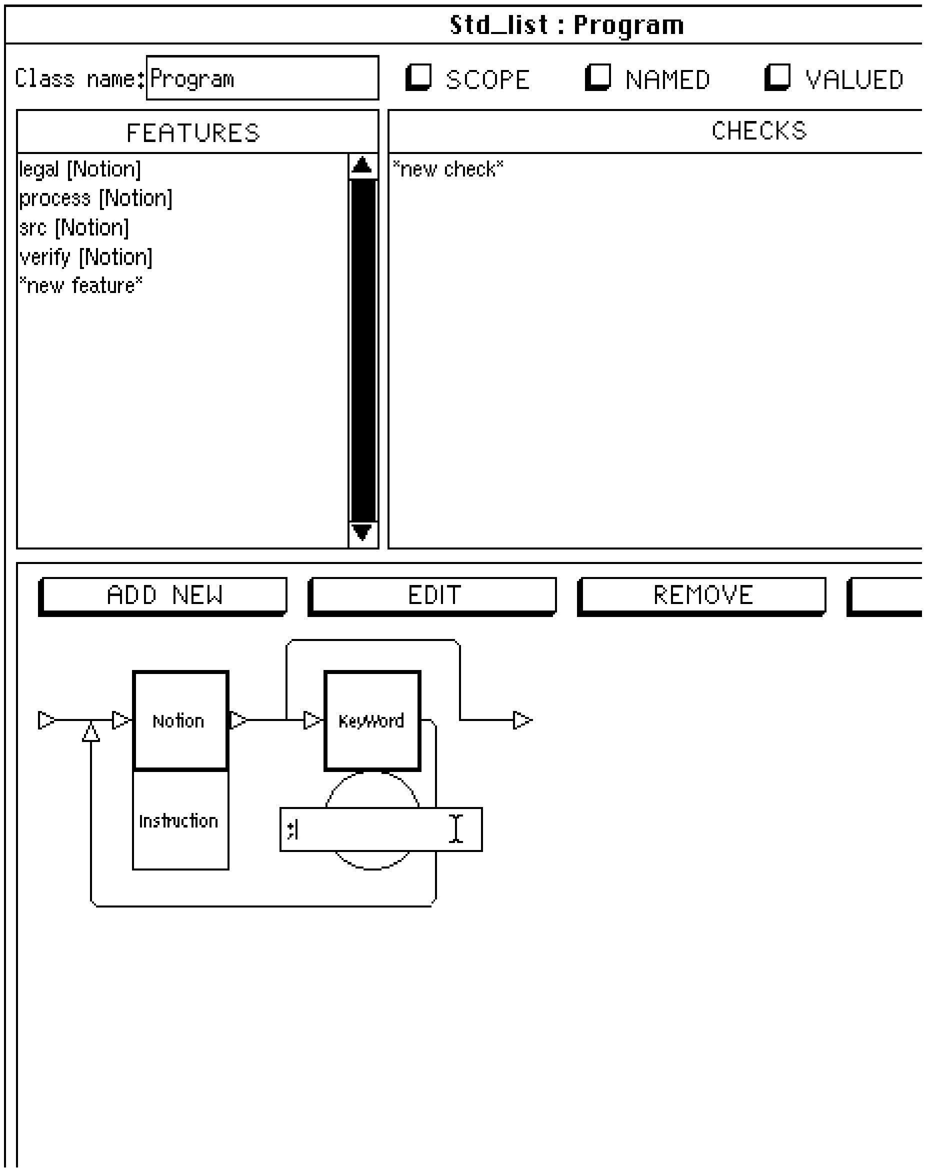

Fig.7. Part of the view shown for giving a subclass of Std_list. The user has just typed in the separator symbol (";").

Summary

This example demonstrates some of the advantages of TaLE, when compared to more conventional language implementation systems.

1) The user need not know any special language implementation paradigm or formalism: we only expect that s/he understands object-oriented programming and the underlying OO language (here Eiffel). Most of the work is done using generally understandable graphical facilities in an interactive environment. The user has the feeling of editing an Eiffel program rather than that of writing a formal specification.

2) If the language to be implemented is reasonably compliant with the existing classes of TaLE, the implementation can be carried out with very little work. In particular, predefined classes can be either directly adopted in the new language (like Pascal_id in the example), or general classes can be specialized for the language by few selections of descendant classes (like using Std_list for Program). The user needs to write code only few lines, and even this is strongly supported by menus.

3) The system is open. Except for the fact that the implementation classes are viewed through a specialized editor, there is nothing special in those classes. The interfaces of

the classes are visible and understandable for the user, and s/he may use them freely in other software. The internal representation of the source is a normal collection of objects that can be associated with arbitrary processing. There are no hidden interpreted system files.

4) The system is incremental. As long as the interface of a class is not changed (e.g. by removing a user-given operation), the editing of a class does not necessitate the reproduction and/or reprocessing of all the other classes, or even the client classes. For instance, syntactic changes can be freely made in the patterns without affecting the other classes. There is no global information about a language that should be updated after each modification. Individual language structures can be tested independently.

4 Basic language model

Class categories and reuse

The TaLE language model is strongly influenced by the requirements motivated by the reuse of language implementation software. In particular, we aim at a hierarchical model in which new subclasses can be defined for general-purpose classes by introducing more and more knowledge about the components of the language structures. This approach leads to different class categories:

1) There are general-purpose classes for representing language concepts without any structural considerations at all. These classes define no components but only purely nonstructural, semantic properties of language structures; examples in TaLE are the universal language construct class and the classes for scope regions, valued structures, and named entities.

2) There are general-purpose classes for representing language structures in which some essential components are known on the abstract level only: i.e., their order and syntactic appearance are not known. Semantic operations of language structures can be thus defined on the basis of the abstract components, without any syntactic considerations, and these operations can be inherited by the concrete language classes introducing language-specific syntactic aspects.

3) There are general-purpose classes for representing language structures in which the order of the appearance of the components is known, although the exact syntactic form of the components or delimiting keywords is not fixed. The order may be necessary because language processing activities may depend on the processing order (e.g. one-pass processing).

4) There are language-specific classes which fix the syntactic form of the language structures they represent, possibly using or redefining operations inherited from general-purpose classes discussed above. This kind of a class can be derived from a general-purpose class by adding all the information necessary for obtaining the concrete syntax of the corresponding language structure.

The conventional object-oriented concepts are tuned for language processing along these lines. As usual, a class defines the features the instances of the class possess; the features are either attributes or operations. In TaLE, an attribute may be defined as a *component*, implying that the attribute refers to an object representing a constituent part of the language structure in question.

A class is either *conceptual* or *structural*. A conceptual class is associated with a (possibly empty) unordered *component set*. The subclasses of a conceptual class can refine the (static) classes of the components, extend the component set, or introduce a pattern for the components (see below). A structural class is associated with a (possibly empty) *pattern*. A pattern is an ordered set of components. The subclasses of a structural class can refine the (static) classes of the components in the pattern, but they cannot introduce any more new components or otherwise change the pattern. For any class, the subclasses may introduce new non-component attributes or operations, or redefine existing operations in the usual way. Since a structural class contains more information about the components than a conceptual one, a structural class can inherit a conceptual class but not vice versa. A pattern is a final template for the actual syntactic form; hence it is possible to inherit at most one structural class. Otherwise multiple inheritance is allowed and often necessary. In the list of class categories given above, the two first categories are obtained as conceptual classes (in the first category the component set is empty), and the remaining two categories are obtained as structural classes.

For technical reasons, we introduce an additional class category, a *terminal class*. A terminal class is a special kind of a structural class, named with a string denotation (e.g. "BEGIN"). We define the pattern of a terminal class as the sequence of the character symbols in this string. All the terminal classes are subclasses of a predefined class Keyword.

As a (strongly simplified) example of the idea of reuse and hierarchical abstraction levels in TaLE, consider the derivation of a class for conditional statements in a particular language. A possible hierarchy is given in the following table, where a class inherits the class above. In this hierarchy, only the last class is to be given by the language implementer; all other classes are assumed to be provided by the general-purpose library of TaLE. We will denote component sets as {C1, ..., Ck} and patterns as [C1, ..., Ck], where Ci ($1 \le i \le k$) are the classes of the components.

class	*category*	*component set/pattern*	*(re)defines operations*
Notion	conceptual		process (virtual) check (virtual)
Cond	conceptual	{Valued, Notion}	process check
FullCond	conceptual	{Valued, Notion, Notion}	process
StdIfStat	structural	[Keyword Valued Keyword Notion Keyword Notion Keyword]	
MyIfStat	structural	["IF" MyExpr "THEN" MyStat "ELSE" MyStat "END"]	

Here Notion describes the universal features of all language constructs, including two virtual operations: "process" for activating the dynamic behaviour of the construct, and "check" for validating its legality. Cond gives two components, one for the condition and one for the construct to be activated if the condition evaluates to true. The class of the former is Valued; we assume this is a class defining features required for constructs associated with a run-time value (this is actually a predefined class in TaLE). The class of the latter is Notion, implying that any language construct can play the role of the component. Cond could define the implementation of "process" roughly as follows:

```
process is
   do
      c1.process;
      if c1.val.is_true then
         c2.process
      end
   end;
```

where c1 refers to the Valued component and c2 refers to the Notion component. Function "is_true", defined for Valued objects, returns true if the value represents the truth value "true". Cond also defines the implementation of "check": this operation verifies that the type of the Valued component (given by another attribute of Valued) is Boolean.

FullCond extends the primitive conditionality of Cond by an else-part: it introduces a new component for the construct to be activated in the false-case. It must also redefine the "process" operation in the obvious way, whereas the "check" operation remains valid.

The rest is straightforward, since only some syntactic issues must be fixed. First, we give class StdIfStat which introduces a pattern for the components of FullCond: now the components must appear in a particular order, with certain positions for keywords. However, there is still a considerable amount of flexibility because the component classes are not refined (and the keywords are not chosen). This refinement is done in the last, language-specific step: class MyIfStat is defined as a refinement of StdIfStat where the component classes (including the keywords) are refined with language-specific descendant classes. We assumed here that MyExpr and MyStat are language-specific classes defined as descendants of Valued and Notion, respectively. In the example language in Section 3, class Program was constructed in this way.

For a conceptual class, the graphical view shows the components in the pattern section, but without arrows. When a structural subclass is introduced for a conceptual one with components, the inherited components are shown in the pattern section with thick borders indicating that they cannot be edited. The user must then construct the actual pattern, and associate each inherited abstract component with a concrete one in the pattern with mouse. In the example language, class Output was constructed in this way.

Syntactic model

Syntactic derivation is accomplished by two mechanisms: pattern replacement and descendant class replacement. The latter is a consequence of inclusion polymorphism in object-oriented languages: if B is a descendant class of A, any context requiring an A object can also manage with a B object. As usual in object-oriented versions of context-free grammars, we describe syntactic alternation using this mechanism. For instance, in a particular language we might have a conceptual class Statement whose subclasses are AssignmentStat and WhileStat; therefore Statement can be replaced by AssignmentStat or WhileStat in syntactic derivation. Formally, we define an object-oriented context-free grammar as follows:

An *object-oriented context-free grammar* (OO-CFG) is a 6-tuple (S,C,T,P,H,Z), where S is the set of structural nonterminal symbols, C is the set of conceptual nonterminal symbols, T is the set of terminal symbols, P is the set of productions of the form $N\text{->}w$ ($N\in S$, $w\in(S\cup C\cup T)^*$), 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 $(A\text{->}w)\in P$ for some w; ($(A\text{->}w)\in P$ and $(A\text{->}w')\in P$) implies $w=w'$ (there is exactly one production for each structural nonterminal);
3) $A>+A$ holds for no $A\in C$ (no cyclic hierarchies);
4) for each $A\in(S\cup C)$: $Z=>^*uAv=>^+w$, 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.

To make the syntactic analysis feasible, we require that those classes participating in syntactic derivation are sufficiently well-defined: the application of the two derivation mechanisms must eventually yield a sequence of terminal strings. We say a class is *complete* if it satisfies this requirement; informally, a complete class is one which defines a full concrete syntax for the structure it represents. Usually only language-specific classes are complete.

Let C be a complete class. The language generated by C is defined as $L(C) = \{w \mid C =>^+w$, w contains no more class names$\}$. Let $x\in L(C)$. Consider a derivation chain for x: $C=w_1=>w_2=> \dots =>w_k=x$ ($k\geq 1$). The *representation of x with respect to* C is the collection of objects obtained by the following algorithm. The algorithm associates an object with each class occurrence in the derivation chain as follows:

For i:= k downto 2 do

a) if $w_{i-1}=>w_i$ applies pattern replacement $zAu => zB_1\dots B_nu$ ($n\geq 0$), where A is a terminal class, do nothing;

b) if $w_{i-1}=>w_i$ applies pattern replacement $zAu => zB_1\dots B_nu$ ($n\geq 0$), where A is a non-terminal class, then:

   1) create an instance of A,
   2) associate the created object with this occurrence of A,
   3) let each component attribute of this object refer to the object associated with the corresponding B_i;

c) if $w_{i-1} \Rightarrow w_i$ applies subclass replacement zAu => zBu, let the object associated with B be the object associated with A.

As a result, there will be an object associated with C; this is the root object of the representation of x. The created collection of objects forms a tree structure starting from the root object, according to the component attributes.

We employ an extended LL(1) parsing technique in which the required starter and follower symbols are computed dynamically, for the needs of a particular input text (see Section 5). This method implies syntactic restrictions which are somewhat weaker than in pure LL(1) parsing; in particular, the method allows dangling-else type ambiguities and the use of symbol table information to resolve parsing conflicts.

Only complete classes need parsing support, but since the code for a class must not depend on the definition of other classes (because of the incrementality requirement), the parsing code must be generated for all potentially complete classes; in this way the code of a class need not be revised even if the class changes its status from complete to non-complete or vice versa, due to modifications in some other classes. However, the system refuses to test a language represented by a non-complete class.

5 Implementation

Syntactic analysis

A major problem in designing the syntactic analysis in TaLE has been to guarantee incremental language development: the classes taking part in the language implementation should be highly independent so that they can be put together in various combinations to form different language implementations, that they can be created and modified without making other classes invalid[1], and that they can be tested independently. A central idea of TaLE is that the concept of a "language" in the traditional sense is eliminated; instead, the basic unit is a language feature, represented by a single class or a collection of classes. This philosophy leads to many unconventional decisions in the implementation model. A detailed account of the TaLE implementation model is given in [JäK93].

Since we want to allow "static" user-defined actions at certain pattern positions (see Section 3), the natural choice is to use top-down deterministic parsing. To satisfy the requirement of incremental analyzer construction we have used the idea of lazy recursive descent parsing presented in [Kos90], and modified it to fit object-oriented context-free grammars. This modification, so-called metaobject-directed parsing, has been discussed in detail in [KoV92b].

The basic idea of syntactic analysis in TaLE is to distribute the "global" information needed for syntactic analysis into objects representing the language classes. This information is computed dynamically, so that classes become statically independent of each other (as far as parsing is concerned). For efficiency, the information is computed in a lazy fashion only when it is really needed for a particular input. The objects that carry this information are called *metaobjects* because they represent "metainformation" (i.e. nonterminals) and because their task is to create instances of the actual language classes, on the basis of the input stream. Each actual language class has a corresponding metaobject that creates instances of this class; the metaobject is in turn an instance of the *metaclass* of the actual language class.

When created, each metaobject constructs a so-called *starter list* for itself; this is a list of the possible starters of the corresponding language structure such that each starter is associated with the metaobject that will be responsible for creating the next instance if the current input matches with this starter. Further, the metaobject must find out whether it can produce the empty string or not. Slightly simplifying, the starter list of the metaobject of a conceptual nonterminal is the union of the starter lists of the metaobjects of its subclasses; the starter list of the metaobject of a structural nonterminal is obtained by taking the union of the starter lists of the metaobjects of the component nonterminals from left to right until a component not producing empty appears, and by changing the

---

[1]Actually, in Eiffel it is not possible to achieve full incrementality due to the so-called problem of unexpected classes [KoV92]: if a new class is introduced, or if an existing class is removed, the immediate superclass has to be revised as well.

second item (see below) of each element in the list to refer to the metaobject itself. Only those metaobjects are created which are needed for the particular input.

A characteristic feature of metaobject-directed parsing is that there is no conventional global scanner: such a scanner would ruin the incrementality requirement. Scanning is performed in two places: when trying to match the starters in the starter list with the current input, a metaobject must effectively perform look-ahead by scanning the starter symbols; on the other hand, terminal symbols appearing in the middle of patterns are scanned directly as individual strings. Hence the scanning is distributed: only those terminal symbols are considered which are really possible in the syntactic context. Since there are usually very few such symbols in a given context (often exactly one), the scanning can be made fast; this compensates some of the overhead of the dynamic starter list construction.

Metaobject-directed deterministic top-down parsing works as follows (see Fig. 8). Assume that the necessary metaobjects have been created, and that we are expecting an instance of nonterminal A in the input stream. The metaobject of A gets a request to create an A. It consults its starter list to find the appropriate structural descendant class of A (say B) that matches with the current input. The metaobject of A then delegates the creation task to the metaobject of B which creates an instance of B. During the initialization code of B the metaobjects of the components appearing in the pattern of B are in turn asked to create the corresponding instances, and the process continues. If A is structural, it need not consult its starter list because a structural class has exactly one pattern which it can directly apply.

In a simplified form a starter list is a list of pairs (s, r), where s is a reference to the metaobject of a starter structure and r is a reference to the metaobject that will be responsible of creating the appropriate instance in the case of that particular starter. Each metaobject has two essential operations: one for creating the appropriate instance (make) and one for performing look-ahead (that is, answering the question: "are you coming next in the input?"). When consulting its starter list, the metaobject applies the look-ahead operation to the s components of the pairs until it finds a success, and delegates then the creation job to the metaobject indicated by the corresponding r component. The look-ahead operation is implemented in class-specific ways for classes representing various token categories; for instance, for an identifier it is sufficient to verify in the look-ahead operation that the next input character is a letter (assuming that the possible keywords have been already checked). The default implementation of look-ahead simply tries to analyze the whole structure and then restores the input pointer to the original position.

Usually the first item s in a starter pair (s,r) is a reference to a metaobject of a terminal symbol; hence calling its look-ahead operation has the effect that the terminal symbol is matched against the input; this corresponds to rather normal decision making in descent style parsing. However, in principle the first item s can denote any metaobject: the "starter symbol" may also be a normal nonterminal, in which case - recalling the default look-ahead implementation - the parser tries to analyze the whole nonterminal before deciding which alternative it takes. For a nonterminal that wants to act as such a "starter symbol", it is sufficient that its metaobject has a starter list with a single element (t,t), where t denotes the metaobject itself. This is exactly the effect of the TOKEN button in the interface (see Section 3). In this way the user can define arbitrary "terminal" structures that are fully taken into account in LL(1) look-ahead. In principle this would also allow significant weakening of the LL(1) condition by applying this to non-token structures as well, but so far this possibility is not included in the system.

Note that the use of metaobjects unify the treatment of different classes appearing in the pattern: independently of the kind of the class (structural or conceptual), the metaobject of the class is asked to create the appropriate instance of the actual structural class. Hence the clients of a class need not know the category of the class they are using.

If a conceptual nonterminal may produce the empty string, its metaobject has an attribute containing a reference to the structural metaobject (say, s) through which the empty string can be produced (there can be only one such metaobject in LL(1) grammars). If the conceptual metaobject does not find a match when consulting its starter list, but the metaobject knows that an empty string may also be produced by the corresponding nonterminal, it assumes that in this situation the empty alternative must be the correct one, and delegates the make operation to s. This decision cannot be wrong in the case of a correct input; in the case of an incorrect input the only drawback is a delayed and possibly strange error message.

[Figure 8 is omitted from the electronically distrubuted version of the report.]

Fig. 8. Metaobject-directed parsing.

Since the analyzer always prefers a non-empty choice to an empty one, and selects the empty choice only as a default move, it is capable of solving LL(1) parsing conflicts which arise from the fact that the starter of a nonterminal which may produce empty is the same as its follower (provided that the correct decision in such a situation is to prefer the non-empty choice). A typical practical example is the dangling else problem.

The distributed scanning of metaobject-directed parsing causes a particular problem. Since a metaobject knows only those terminals that are its starter symbols, it can in some cases find a "wrong" match. This can happen if the nonterminal in question may produce the empty string, and a prefix of some of the possible follower symbols is structurally equivalent with a starter symbol: then the metaobject would make the false decision that a non-empty alternative must be chosen. Although this situation is not very common, it does occur in practical languages. We have solved this problem by maintaining a top-down parse stack consisting of references to the metaobjects whose nonterminal instances are expected to appear in the rest of the input. If a metaobject is going to make a decision that could be wrong in the above sense, it first consults the next metaobject(s) in the parse stack to make sure that a follower symbol does not collide with the selected starter. The cost of using the additional parse stack is discussed below.

The language processors produced by TaLE are relatively slow: a typical processing speed is 50 lines per second (including reading the source, scanning, parsing, and the construction of the internal object form). However, the slowness originates from Eiffel 2.3 rather than from the techniques we have used. To demonstrate this, we implemented by hand an analyzer for a Pascal subset ("Mini-Pascal" from [WeM80]) following the techniques discussed above but using Borland C++ in a 40 MHz 386 PC. To evaluate the cost of the additional parse stack, we implemented both a complete version with the stack (MDP) and a reduced one without the stack (MDPred). For comparison, we also implemented a traditional recursive descent analyzer with global scanner and parser objects (TRD); this analyzer represents the optimal case as far as efficiency is concerned. We measured the processor time consumption of source programs of varying length, obtaining Figure 9.

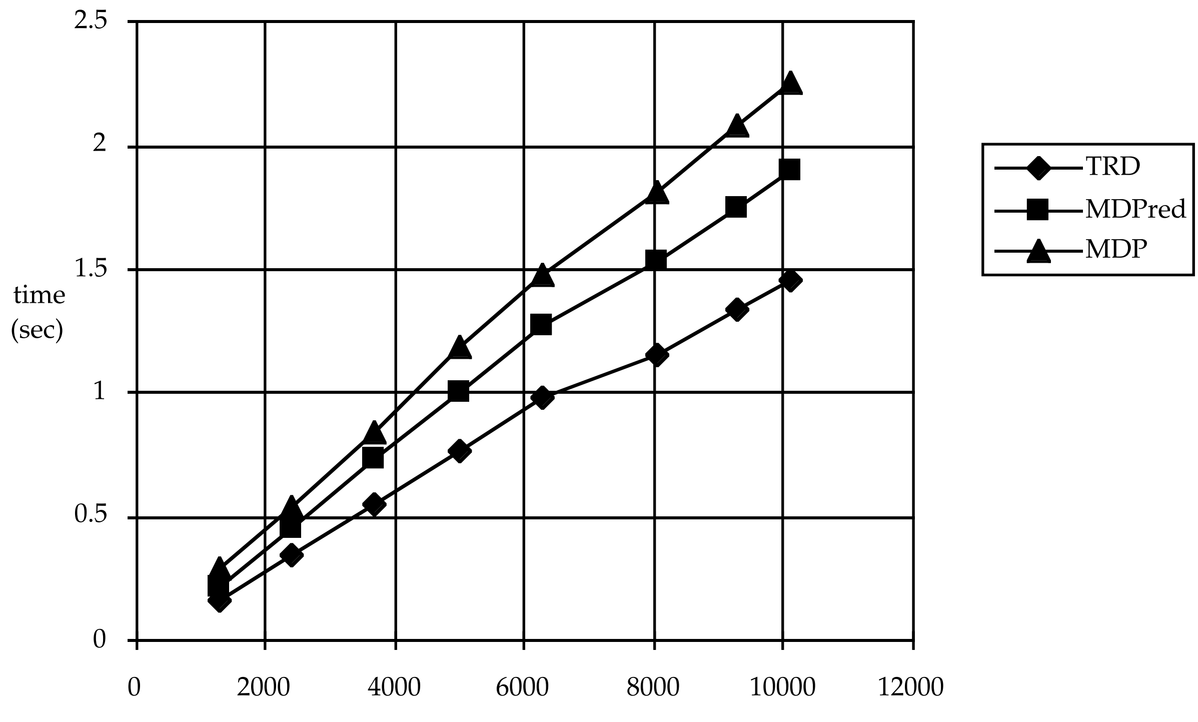

Fig. 9. Comparison of processing times.

As can be expected, the time usage is linear for all the analyzers. When compared to the TRD analyzer, the MDP analyzer is about 50% slower. The time cost of the additional parse stack is about 15%. Besides the use of the parse stack, the difference between the MDP analyzer and the TRD analyzer is due to the creation of the metaobjects with their starter lists, and to the heavy use of virtual functions (which are not needed at all in the TRD case). Yet, the absolute processing speed of the MDP analyzer is about 4000 tokens per second which is sufficiently good for any practical purposes.

Name analysis

Metaobject-directed analysis allows a simple technique for integrating parsing with name analysis, making it possible to use context-sensitive information to resolve parsing conflicts. For instance, suppose that Statement has descendant classes AssignmentStat and ProcCall both beginning with an identifier. Then the starter list of the metaobject of Statement has the pairs (id,as) and (id,pc), where id is a reference to the metaobject of Pascal_id (an identifier), as is a reference to the metaobject of AssignmentStat, and pc is a reference to the metaobject of ProcCall. If there is an identifier coming next in the input, the look-ahead will succeed for both pairs which is not allowed. However, the language implementer can solve the problem by qualifying (see Fig. 6) the leading identifiers.

Assume that for AssignmentStat the leading identifier is qualified with class Variable and for the ProcCall with class Procedure. Then the items in the starter list are triples (id, va, as) and (id, pr, pc), where va is a reference to the metaobject of Variable and pr is a reference to the metaobject of Procedure. When such a triple item is consulted and the look-ahead on the first component (id) succeeds, the metaobject given by this component is additionally asked to actually create the instance with the make operation. In the case of an identifier starter this means the scanning of the whole identifier. The resulting (identifier) object has a dedicated string-valued attribute (key) that gives a key of an object residing in the object base; by default, this attribute contains the string representation of the object itself, as appearing in the source text. Object base look-up is then performed on the basis of this key, and if an object is found having this key and agreeing with the given qualification class, the look-ahead is considered successful and the make command is issued to the metaobject denoted by the third component(as or pc).

External analyzers

The distributed parsing philosophy of TaLE makes it possible to mix different analysis methods easily: each class is supposed to take care of itself only, and it can freely choose the method it considers the best. The only requirement is that the metaclass acting as an interface for other TaLE classes is constructed according to the general model; i.e. this class must provide the standard services for look-ahead and for creating the language structure in question. However, the latter operation may simply call an external parser to do the job. We have applied this technique in the implementation of the standard expressions: structures which are defined as subclasses of Std_expression are actually parsed using the fast and compact method presented in [Han85], instead of the general (less efficient) lazy parsing method. The use of such external analyzers is supported in TaLE through a special class whose subclasses can be defined under a view that allows the explicit giving of the starter symbols.

6 Concluding remarks

TaLE is a step toward “granular languages”; i.e. languages that are composed of relatively independent, replaceable, partly standardized units. As different application areas become more mature and systematic, there will be a growing demand for special-purpose languages for those areas, and we anticipate that these languages make more and more use of standard components: individualism becomes too expensive. We expect that different TaLE class libraries will be developed for different application domains, so that the language development can be done (re)using concepts that are already near to the language. In the simplest case, each language implementation carried out in TaLE is in fact a specialized class library that can be utilized e.g. in the development of the next generation of the language.

Typical applications of TaLE are the implementations of application-oriented languages (say, process control languages, query languages, robot control languages, document description languages etc.), processing of intermediate data (or object) descriptions, analysis of ASCII files, generation of software from its specification, interpreting user interface specifications, experimental implementations of programming languages, source-to-source translations etc. TaLE is not intended for the development of production-quality compilers of general-purpose programming languages.

TaLE demonstrates the benefits of the object-oriented programming paradigm in (computer-aided) language implementation. A unified language model can be developed such that the model can be followed both in the user interface and in the actual software; this model implies an abstract internal representation of the source. General, language-independent notions can be presented as reusable classes, specialized through subclassing. The information concerning a language can be distributed into more or less independent classes, allowing thus the incremental development of a language implementation. Through dynamic binding the classes can be individually implemented without losing the simplicity of the general model. Hence TaLE draws heavily on the properties of an object-oriented base language; indeed, the use of an object-oriented base language seems to be an essential requirement for a metalanguageless approach to automated language implementation.

TaLE opens up new research directions particularly in the areas of object-oriented language implementation techniques and graphical language specification. Work in these directions continues. The basic problem in the automation of language implementation is to understand the nature of the tool that is really needed by practical programmers. TaLE tries to bring computer-aided language implementation and software engineering together, emphasizing issues like code reuse, incremental software development, object-oriented software development, and graphical specifications. Although we feel that TaLE is a major step in the right direction, more real-life experiences are needed to evaluate the advantages and shortcomings of the system. We are currently applying TaLE to industrial language implementation problems.

Acknowledgements

This work has been financed by the Academy of Finland (grant 1061120), and by the University of Tampere. We have received significant support from our fellow researchers Jukka Paakki and Juha Vihavainen. We wish to thank the following students who have programmed parts of the system: Maarit Niittymäki, Annika Knuuttila, Janne Lahti, Marianne Setälä, Jari Torkkola, Hannu Vainio-Mattila, and Toivo Venäläinen. The C++ experiment discussed in Section 5 was carried out by Jaana Isaksson, Mikko Laitamäki, Tarja Systä, Anssi Takala and Jorma Törnblom.

References

[AMH90] Aksit M., Mostert R., Haverkort B.: Compiler Generation Based on Grammar Inheritance. Mem Informatica 90-07, Department of Computer Science, University of Twente, February 1990.

[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.

[Ben86] Bentley J.: Little Languages. Programming Pearls, CACM 29,8 (August 1986), 711-721.

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

[FJS86] Feiler P., Jalili F., Schlichter J.: An Interactive Prototyping Environment for Language Design. In: Proc. of the 19th Annual Hawaii International Conference on System Sciences, 1986.

[Gol84] Goldberg A.: Smalltalk-80: The Interactive Programming Environment. Addison-Wesley, Reading, Massachusetts, 1984.

[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.

[GHK88] 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.

[Han85] Hanson D.R.: Compact Recursive-Descent Parsing of Expressions. Software Practice & Experience 15, 12 (1985), 1205-1212.

[HKR89] Heering J., Klint P., Rekers J.: Incremental Generation of Parsers. In: Proc. of ACM Sigplan '89 Conference on Programming Language Design and Implementation, Portland, Oregon. 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.: Language Implementation Model in TaLE. Report A-1993-1, Department of Computer Science, University of Tampere, 1993.

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

[KoP87] Koskimies K., Paakki J.: TOOLS - A Unifying Approach to Object-Oriented Language Implementation. In: Proc. of ACM Sigplan '87 Symposium on Interpreters and Interpretive Techniques, Sigplan Notices 22,7 (July 1987), 153-164.

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

[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 Construction with Metaobjects. Report A-1992-5, Department of Computer Science, University of Tampere, 1992.

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

[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.

[ReM85] Rechenberg P., Mössenböck H.: Ein Compiler-generator für Mikrocomputer. Hanser Verlag, München, Wien, 1985.

[ShK90] Shinoda Y., Katayama T.: Object-Oriented Extension of Attribute Grammars and its Implementation. In: Workshop on Attribute Grammars and Their Applications, P. Deransart, M. Jourdan (eds.), Paris, LNCS 461, Springer-Verlag, 177-191.

[WeM80] Welsh J., McKeag M.: Structured System Programming. Prentice-Hall 1980.

[Win87] Winograd T. A.: Muir: A Tool for Language Design. Report STAN-CS-87-1159, Department of Computer Science, Stanford University, May 1987.

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