A Flexible Object Invocation Language based on Object-Oriented Language Definition

Mark Evered
Axel Schmolitzky
(University of Ulm)

Michael Kölling
(University of Sydney)

Contact Address:	Dr. M. Evered Abteilung Rechnerstrukturen Universität Ulm 89069 Ulm Germany
Telephone:	+49 731 5024179
FAX	+49 731 5024182
Email:	markev@informatik.uni-ulm.de

Abstract:

The objects accessible by a user at a particular time in an object-oriented system form a working environment. The invocation language via which this environment is viewed and manipulated can be regarded as semantically flexible since the set of operations available to the user is the union of the methods of all currently visible classes. In this paper we present a language system in which class definitions are extended to include appropriate syntax rules for the methods and which is consequently also syntactically flexible. The requirements for such a system both from the point of view of a user and with regard to an efficient implementation are discussed. An implementation of such a flexible invocation language within a persistent object-based operating system is then presented.

1. Introduction

An interactive language is a means of communication between a user and a software system. The first interactive languages were command languages. The software system involved was an operating system and the commands were concerned with invoking operating system services to manipulate files, programs, devices and processes. More generally, however, the term 'invocation language' can be associated with any kind of interactive software system which allows a user to specify an action and responds by invoking a program or object to perform that action or by indicating an error in the input. The action may be to display information regarding the state of the system or to alter its state. The more general use of the term is particularly valid in the case of object-oriented systems where there is no clear distinction between operating system objects, database objects and application objects.

Most research on interactive systems in recent years has concentrated on graphical user interfaces. This research has also made it clear, however, that textual interfaces or invocation languages are the more favourable alternative in many situations (Balzert, 1988). Even with graphical interfaces there is a need for some actions to be specified in the form of textual input. In this paper we concentrate on textual languages, in the traditional sense, although some of the ideas are equally applicable to graphical environments.

Two aspects of object invocation languages which have become increasingly important are *extensibility* and *adaptability*:

- We define extensibility as the ability of a language to be extended to new domains and to express adequately the new actions offered by the system.
- We define adaptability as the ability to offer commands in a form suitable to a particular domain and to the requirements and preferences of a particular user.

While early command languages such as JCL (IBM, 1966) were restricted to a fixed set of system commands, the command languages of later systems such as Unix (Ritchie and Thomson, 1974) can be extended by the incorporation of new programs and the use of parameterised command files. The 'alias' mechanism of Unix offers a simple kind of adaptability. The first word of a command line is matched against a user-specific list of aliases and, if found, is replaced by a particular character string. In this way a user can give a new name to a command or to a command combined with particular options and parameters. These features are, however, very limited. The basic form of a command always remains the same – the name of the command followed by a number of space-separated character strings representing the parameters. In such systems the semantics of the command language is extensible but the syntax is not. So, for example, the command language semantics could be extended by incorporating into the system a program for adding integers but this must then be invoked using the prefix syntax:

```
int_add 3 2
```

rather than the more intuitive infix form:

```
3+2
```

A flexible syntax is equally important in achieving the goal of adaptability. The natural notation for one application area will not generally be the same as for another; while one user may prefer a concise, cryptic syntax, another may favour a verbose, self-explanatory style. Ideally an invocation language will adapt to optimally suit the domain in which a user is currently working. This kind of syntactic extensibility and adaptability is not possible given the rigid form of current invocation languages.

The subject of extensible syntax has been taken up repeatedly throughout the history of computing. As early as the 1960's suggestions were made as to how special macros could be used to make programming languages syntactically extensible. Leavenworth (1966) proposed the use of 'syntax macros' which could map a given syntax on to either a statement or an expression. These macros always had a macro name as the first item in the syntactic pattern, thus severely restricting the expressive power. More recently, a mechanism has been suggested (Cardelli et al., 1993) whereby layers of syntactic extensions and restrictions can be built above a basis*** λ-calculus language to define LL(1) database languages. All of the these attempts to define an extensible language face two main problems. One problem is in defining a mechanism which is easy to use and manage. The other major problem is in achieving an efficient implementation of the extension mechanism.

Standish (1975) has classified three dimensions of extensibility:

- 'Paraphrasing' is restatement in another form. This is supplied by most languages via macros, procedure definitions and type definitions.
- 'Orthophrasing' refers to an extension beyond the original capabilities of the language, for example access to file systems and low-level I/O operations. This is achieved in object-oriented systems by treating system objects and application objects uniformly.
- 'Metaphrasing' is an extension which allows an old expression of the language to be interpreted in a new way. Ada (ANSI, 1983) for example, allows the infix operators '+', '–' etc. to be overloaded and used for new user-defined procedures. The Galaxy language (Beetem and Beetem, 1989) allows special prefixes and suffixes as well as the usual arithmetic symbols to define the pattern by which a user-defined function or macro is to be invoked. C++ (Stroustrup, 1989) allows a large set of special symbols

to be used as unary or binary operators for the methods of new object classes. Smalltalk-80 (Goldberg and Robson, 1989) has a similar mechanism and can in fact also be seen as an object invocation language, but here also a rigid syntax (object name, method name, parameters) is maintained. The flexibility consists only in permitting special symbols as method names.

A more adaptable syntax for object invocation languages could allow an arbitrary ordering of operators and operands and could interpret each component in the context of the whole input line. A more powerful mechanism than those in current language systems is required if these languages are to be as extensible and adaptable syntactically as they are semantically.

The next section describes a new approach to defining languages which allows such flexibility and which is based on the objects themselves. The subsequent section examines the specific requirements for this approach when applied to object invocation languages. Finally we present a system which efficiently implements the ideas in the context of a persistent object-oriented architecture.

2. Object-oriented Language Definition

The advantages of object-oriented software design are well recognised. Internal object data can be hidden behind a well defined functional interface. This in turn can enhance reusability and error localisation and minimise interactions and complexity.

In an object-oriented environment the total functionality of the system is represented by the sum of all methods of all callable objects. An object invocation language can offer each of these methods as a command. In fact the semantics of the user language at a particular time is a representation of the sum of the functionality of the objects which can be invoked at that time. As new objects are added to a system, or as the working environment of a user changes, the set of objects in use and, consequently, the semantics of the invocation language are changed.

The user language of a system can thus be seen as a *dynamic language* defined by the objects available to a user at a particular time. In order to offer the kind of extensibility and adaptability described above, the invocation language must have a dynamic syntax as well as dynamic semantics[1]. This can be achieved by including in an object definition not only a functional description of its methods, but also a description of the syntax by which the methods are to be invoked. The sum of the syntax of all available objects then defines a *dynamic grammar* for the language.

The syntax of programming languages is generally defined by a context-free grammar. In object-oriented language definition (Evered, 1993), the syntax of a language can be defined by

[1] By 'dynamic' we mean syntax and semantics which vary over time, not 'dynamic' in the sense of dynamic binding.

a distributed context-free grammar, the rules of which are spread over the objects comprising the system. This can be visualised as shown in Fig. 1.

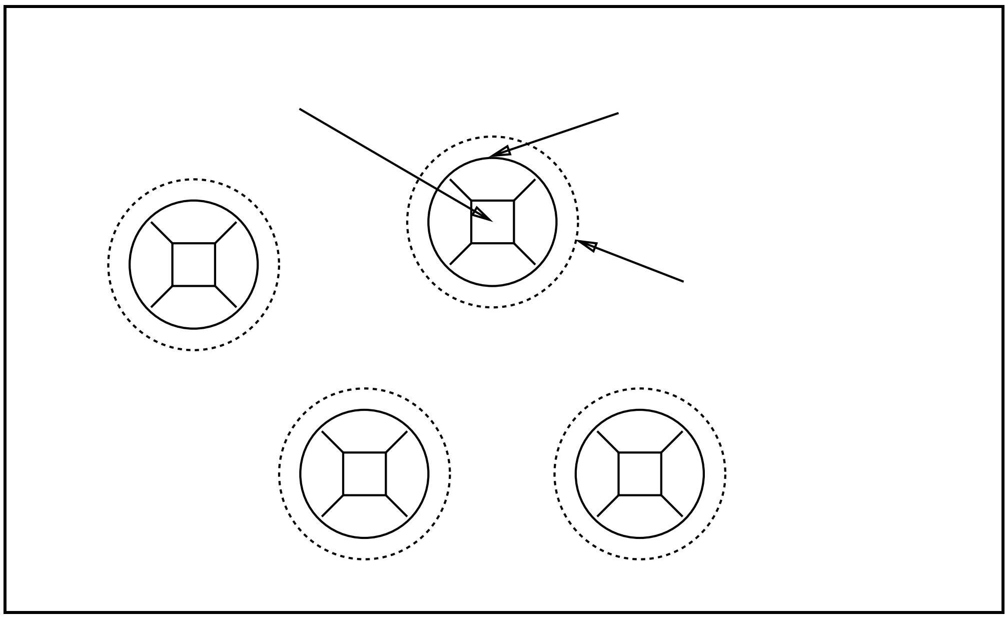

Fig. 1: Objects with syntactic shells

Encapsulating the functional interface of each object is a shell which interprets the syntax to be used in accessing this object. This is similar to the distinction in Unix (Ritchie and Thomson, 1974) between the system calls which offer a functional interface to the system and a command shell which describes the syntax by which this functionality can be invoked. Here, however, in accordance with the object-oriented philosophy, the functionality and the shell are distributed among the objects.

The language resulting from the object methods and object syntax rules may be called the external or user language. In order to be executed the sentences of this language must be transformed into the internal language of the system. In object-oriented systems this internal language consists basically of calls to object methods. The syntax rules of an object are therefore not simply context-free grammar rules for parsing a sentence, but also contain a transformation part which maps a parsed section of the external language on to the appropriate expression of the internal language. So, for example, an object implementing integer arithmetic could contain a rule like:

```
int_expr: int '+' int --> 'int_arith.add #1 #3'
```

which, together with other rules, would map the statement 3+2 on to a call to the 'add' method of the object 'int_arith'.

As mentioned above, some fundamental problems are encountered in proposals for syntax extensibilty. One problem is the manageability of the large number of individual syntax rules necessary to define a useful language. A further difficulty is the need for a compromise between flexibility and efficiency. In order to remain relatively efficient, most systems impose strong explicit restrictions on the kinds of syntax rules allowed and implicit restrictions on the frequency with which rules may be added or changed thus making them unsuitable for dynamic invocation languages. A further problem is the possibility of ambiguities in the language. An object-oriented approach to syntax extension alleviates these problems since:

- the syntax rules are made more manageable by being grouped and associated with the classes to which they belong.
- the association of syntax rules with classes makes it easier and more natural to formulate type-specific rules and overloading of syntax patterns.
- only the rules of 'currently relevant' classes (the classes of currently visible objects) need be considered. This increases efficiency and reduces the risk of ambiguity.

The details of this approach depend on the specific requirements for a particular language system. In the following section we discuss more fully the requirements for a general purpose object invocation language in a persistent object-oriented environment.

3. Language Requirements

In order to achieve the goal of providing a high degree of extensibility and adaptability the transformation rules must be able to express more powerful transformation mechanisms than the ones known from conventional systems.

Below we list a number of transformation constructs that are particularly useful in a general purpose language. Using these constructs the internal language can be adapted to the particular needs of a certain user domain.

For each construct we give an example of a command a user might want to use and (after the symbol -->) the equivalent representation of the command in the internal syntax. The transformation system must offer the possibility of defining rules which specify the corresponding transformations.

3.1 Abbreviations

Abbreviation mechanisms are used to give a new name to a command or command sequence (possibly including parameter values). For example, the user should be able to write a rule that causes the following transformation:

```
pc myfile --> 'pascomp.compile -optimise "myfile"'
```

Abbreviation mechanisms are supported by most common command language interpreters.

3.2 Change of the order of elements

As mentioned before the user should be able to specify transformations that change the order of elements (i.e. the command name, options and parameters) in an input line. This eliminates the common restriction that the first word of the input must be the command name. For example, it should be possible to write rules that carry out the following transformations:

```
x:=3  --> variable_manager.assign "x" 3
pascomp? -->  help_manager.man "pascomp"
```

3.3 Defaults

Defaults for parameters should be expressible in the rules. For example:

```
inc a 3  --> int_lib.add "a" 3
inc a --> int_lib.add "a" 1
```

3.4 Context Sensitivity

The meaning of all elements of the input line in the current domain should be useable as a condition for applying transformation rules. The most common example for this is the usage of type dependency. The type of literals, variables and expressions as well as the class of objects can be considered. This allows the appropriate routine to be called for similar (i.e. equally named) functions with different types of parameters. Examples:

```
3+42  --> int_lib.add 3 42
s1+s2 --> string_lib.concat s1 s2
```

It is also possible to use other aspects of significance in the current environment as transformation criteria. For example, a command consisting of a single identifier could be interpreted as an 'rlogin'-command if the identifier is a hostname, or, at the same time, as a 'talk'-command if the identifier is a user name.

3.5 Priority of operations

The user should be able to define the priority of operations. E.g.:

```
2*i+100 --> int_lib.add (int_lib.times 2 i) 100
2+i*100 --> int_lib.add 2 (int_lib.times i 100)
```

In this way complex expressions for data types can be used.

While offering all these kinds of transformation the system must nevertheless be efficient. A runtime environment must be*** developed which executes the transformations quickly. It is envisaged that the whole system would work in an interactive environment where the time of response of every interactive invocation is affected by the time used for the syntax transformation. It is well known that the response time of computer systems is an important factor for the quality of the user interface in terms of software ergonomics. Whether the whole transformation process can be done quickly or not is one of the critical questions in this approach.

4. Further Requirements

As mentioned before, the transformation rules form a context free grammar describing the user language. Using a context free grammar to analyse the whole input line provides a basis for achieving the desired power of the transformation rules.

Parsing and transforming based on context free grammars is well known and commonly used in compilers for programming languages. In the object-oriented definition of invocation languages there are various aspects which cause problems beyond those known from compiler construction. The main differences are that the grammar is not known in advance and that it can change dynamically during the usage of the system. With every addition or deletion of an object into or out of a user's working environment the grammar will change. An algorithm must therefore be provided which handles dynamically changing grammars.

Various problems arise from the fact that the grammar is not developed by a single person who has an overview of the complete set of rules. We have to cope with a situation where different users provide different parts of the grammar without knowing all of the other rules that could be in use at the same time. This leads to several demands on the transformation system.

4.1 Arbitrary Context-Free Grammars

The algorithm must be able to parse arbitrary context free grammars. Because we want a wide group of people to write syntax rules, we do not want to make restrictions on the grammar. Not allowing, for example, certain kinds of recursion can certainly make the parsing algorithm simpler, but would require the user to be capable of detecting and eliminating this kind of recursion, transforming it to another construct. This would exclude a large group of users from the group of rule writers.

4.2 Public and Private Nonterminals

Due to the fact that the grammar will be distributed over several objects, a mechanism has to be provided that on one hand allows a user to define rules for one object without inference with rules for other objects and on the other hand allows object rules of different objects to cooperate.

An example for the latter case could be the start symbol of the grammar. If we assume a nonterminal COMMAND as the start symbol, then every object should at least be able to reference this nonterminal and define new commands.

Object-local rules are similar to local variables in programming languages. These were introduced to ease the task of writing pieces of code that perform an isolated task and do not interfere with variable names in other subroutines.

For these reasons we propose a distinction between *public* and *private* nonterminals. Public nonterminals are accessible in different objects and provide the link between grammar rules which are defined in different objects. Private nonterminals are local to one object. They are only visible within the rules of the defining object.

In the following very simple example public nonterminals are written in uppercase and private nonterminals in lowercase letters.

Object 1	Object 2
COMMAND : 'list' param .	COMMAND : 'print' param .
param : 'cwd' .	param : 'myfile' .

Given these rules, "list cwd" and "print myfile" would be valid commands, whereas "list myfile" and "print cwd" would not.

4.3 Empty Productions

The algorithm should handle rules with empty productions. This exceeds the definition of context free grammars where empty productions are illegal but they can be very useful in defining optional occurrence of elements and defaults.

As an example assume the following rules:

```
COMMAND : 'print' FILENAME copies .
copies :   | INTEGER .
```

where the nonterminal 'copies' has two alternatives, one of which contains an empty production. According to this grammar "print myfile 2" and "print myfile" could be valid commands. In conjunction with transformation directives it is easy to add a default. With the rules:

```
COMMAND : 'print' FILENAME copies .
copies   :     --> '1'
 | INTEGER .
```

the command "print myfile" would be transformed into "print myfile 1".

4.4 Scanners

Various approaches to the task of scanning terminal symbols are possible. The standard method of having a fixed scanner that cuts the input line into tokens before the syntax analysis begins has the advantage of being easy and fast. However it is inadequate here since different rules might rely on different token definitions. A predefined scanner, i.e. predefined tokens, would be a strong restriction to the freedom of syntax definition.

An alternative is to have no scanner at all, relying on characters only and defining everything else in the grammar. This is a very flexible approach but it has the disadvantage of being extremely inefficient in many common cases.

Having a rule based precompiled scanner defined by distributed rules would combine some of the flexibility with a greater efficiency but still has one major problem: different rules depending on a different scanning of the input line could be active *at the same time* and therefore any fixed separation of characters into tokens causes problems.

A solution to this problem is the use of *distributed external scanners* which do not cut the input line prior to the syntax analysis but are called in a top-down manner whenever the grammar derivation process wants to check the input line for the presence of a terminal. Each terminal symbol has a scanner associated with it which tries to scan the symbol at the current position of the input line.

In an object-oriented environment these scanners could be added freely to the system providing a high degree of flexibility and, since it is compiled code, efficiency as well. Moreover since the scanner is defined by a normal piece of code, not only the syntactical aspects of the input line but any computable condition can be a criterion for the scanner to match. This provides a basis for adding semantic conditions to the transformation process. An example of this is the use of a 'username'-scanner and a 'hostname'-scanner. Testing a rule containing a 'username' as a terminal symbol for matching, the scanner would not only scan the name but also check whether the scanned identifier is an existing user's name in the current environment. The two scanners can then distinguish user and host names, even though their syntactic definitions might be equivalent, providing the means for defining the 'rlogin'- and 'talk'-transformations mentioned above.

4.5 Ambiguity

Another result of the fact that a rule writer does not know all active rules is that ambiguous grammars can not be prevented. Since only a small subset of ambiguous definitions can be recognised statically, the parsing algorithm must cope with runtime detection and handling of ambiguous rules.

Solutions to this problem could either provide an algorithm to automatically decide how to solve the ambiguity (including the risk of misinterpretations) or start an interactive dialogue to let the user decide how to continue.

4.6 Error Messages

Generating error messages for a language not known at the time of writing the transformation algorithm is a difficult task. There are two different approaches to this problem: either make the rule writer responsible for providing error messages, or try to let the transformation algorithm generate messages on its own. Either way is however not as simple as in common compiler development. Since we can not be sure of having an LL(k)-grammar, the unexpected ending of a derivation is not necessarily an error. Only if no derivation at all succeeded do we know that something went wrong, but it is then hard to decide the user's intention, since there may have been several partial matches. Even if the rule writer provides messages, the choice of which message to display remains a problem. All error message mechanisms must then rely on heuristics and assumptions in an attempt to choose one or more hopefully helpful messages to display.

4.7 Compiler Generation

Basically there are again two different possibilities to use the syntax rules to carry out the parsing and compilation task. The transformation system could generate a compiler each time the grammar changes and therefore act as a compiler-compiler for arbitrary, ambiguous, context-free grammars. Alternatively, it could act as a rule based *compiler-interpreter* , which means that it executes the tasks of a compiler without actually constructing it. Which of these possibilities is preferable is determined by the time needed to construct the compiler, the gain of performance through the compiler construction and the frequency of grammar changes (i.e. the frequency of constructing a new compiler).

Generally we can say that we expect rather frequent grammar changes and short sentences to be parsed (compared to compilers for programs). Hence, contrary to programming languages, it will probably not be worthwhile constructing the compiler.

The following section describes a system which implements these ideas as a rule based compiler-interpreter for a distributed grammar. The solutions chosen for the problems mentioned above are presented and briefly explained.

5. The MASK System

5.1 The Environment

In this section we introduce our implementation of a command language interpreter (CLI) which uses object-oriented language definition. This implementation (MASK) is based on an existing CLI in an object-oriented environment. The MONADS-CLI (Hitchens, 1989) is the command language interpreter of the operating system of the MONADS-PC (Rosenberg and Abramson, 1985), a research computer which evolved out of the MONADS-Project. The main features of this unusual computer are its large, persistent virtual memory, a two-level capability protection mechanism and its strict architectural support for the information hiding principle. The whole operating system, even the kernel, consists of objects with procedural interfaces. Any action taking place in the system is caused by a call to an interface routine of an object. All objects are instances of classes, which are described in class descriptions. These class descriptions contain information about names of routines, number and type of parameters etc. Most of the operating system is written in MONADS-Pascal, an object-oriented extension of Pascal. Fig. 2 shows a class description for an object which manages simple command language variables.

```
var_manager = class

  function define_var (varname  : string;
                       basetype : string;
                       logtype  : string) : integer;

  function delete_var (varname : string) : boolean;

  procedure list (long : boolean);

  function get_type (    varname  : string;
                     var basetype : string;
                     var logtype  : string) : integer;

  function assign_int (varname : string;
                       val     : integer) : integer;

  function assign_real (varname : string;
                        val     : real) : integer;

  function assign_bool (varname : string;
                        val     : boolean) : integer;

  function assign_modcap (varname : string;
                          val     : modcap) : integer;

  function assign_string (varname : string;
                          val     : string) : integer;

  function get_int_val (    varname : string;
                        var val     : integer) : integer;

  function get_real_val (    varname : string;
                         var val     : real) : integer;

    ...

end; { class var_manager  }
```

Fig. 2: Sample of a MONADS class description: the variable manager

Whenever an object is to be called, the caller must present a *module capability* (modcap) for that object. Module capabilities are the basis of semantic protection in the MONADS system and are protected by the architecture. No call can be performed without a valid capability. Thus the set of capabilities owned by a user defines the objects that user can call. The capabilities can be held in *directories*, which provide a mapping from names to capabilities. Directories are also objects, so a user can insert capabilities for directories into other directories in order to build a tree-like structure.

For efficiency reasons the MONADS-System uses a two-level access strategy for objects. Before an object can be called it must be *opened*. After the required calls have been performed the object can be *closed* again. In this way the effort for the dynamic binding and the checking of the access rights is kept small since it is done only with the open call and not with every call. The set of objects a user can reach via the capabilities in his or her directories are called the *environment* . The set of open objects is the *current context* .

In order to reduce the complexity of system structures the MONADS-System provides a single form for module calls, whether they are interactive calls from the CLI or calls from another object. Thus in the existing CLI on which MASK is based the syntax of an interactive command is:

<object name>.<routine name> { <parameter> }

For every interactively callable object the CLI needs a *template* in which the information about how to call the object is held (i.e. the names of the routines, the number of parameters of a routine, the types of these parameters etc.). This information is extracted from the class descriptions and held in objects of the template class. Every directory entry consists of the capability for the object and the capability for the template.

5.2 The System

Up to this point we can conclude that the semantics of the command language depends on the objects in a user's current context. Thus the existing MONADS-CLI was semantically extensible in the sense of our definition but not syntactically. The first step towards object-oriented command language definition is to provide objects with syntax rules. We decided to place the rules in the class descriptions of the objects. From here they can be added to the information held in a template. Appendix A shows a definition of the 'variable manager' class including syntax rules.

We adhere to the following conventions in the meta-language for the syntax rules. (A formal definition of the meta-language syntax is given as appendix B.)

- A BNF-like notation is used in conjunction with additional transformation directives. A single syntax rule consists of a nonterminal, a production and an optional transformation. Different production-transformation pairs for the same nonterminal are called alternatives and are separated by a '|'.
- Nonterminals are simple identifiers; public nonterminals are written in uppercase and private nonterminals in lowercase letters.

- Terminals are in simple quotes for literals or delimited by slashes for the *identifier scanner* and the *external scanners*. identifier scanner gives the user the possibility of having an identifier recognised in the input. If a type name is included in parentheses then the identifier is subsequently type-checked. External scanners give the user the possibility of invoking hand-written scanners for special purposes (for example to scan an integer literal or a valid user name).
- Transformations are strings enclosed in single quotes. References to elements in the production are given as #<*elem number* >. The #name construct refers to the name of the object from whose template the rule was taken.

For the MASK system the syntax of the existing CLI is the internal language into which the command lines formulated in the external language must be transformed. The external language is defined by the current grammar, i.e. the sum of all rules of the objects in the current context.

Each time an object is interactively opened the central module of the CLI passes the object's template capability and its name (as given in the directory entry) to the main module of the MASK system. If it is the first open object of its class all class rules are added to the current grammar. All occurrences of '#name' are replaced by the name of the instance. If there are new rules for a public nonterminal that already exists in the current grammar the alternatives are simply added to the existing ones.

Each time a command is entered by the user the CLI passes the command line to the MASK module. MASK checks whether it is a valid sentence of the current language by trying to parse and transform it. If the transformation is successful, the transformation result is passed to the CLI as a valid command, otherwise an error is signalled.

5.3 The Transformation Algorithm

The heart of the MASK system is the transformation algorithm, which is based on the parsing algorithm of Earley (1970). We chose Earley's algorithm for several reasons:

- It doesn't need to precompile a given grammar. Instead it directly interprets the grammar during the parsing process. This suits the need for a compiler-interpreter that is obviously given by the dynamic environment in which the grammar can change with every open or close call.
- It uses a top down strategy which is essential for the use of distributed external scanners as described in section 4.
- It makes no restrictions on the form of the grammar since it handles general context free grammars and can furthermore be easily extended to handle even empty productions.

- It has a time bound proportional to $O(n^3)$ (where n is the length of the string being parsed). It has an $O(n^2)$ bound for unambiguous grammars and it runs in linear time on a large class of grammars, which include most practical context-free programming language grammars.

For a better understanding of our extensions to the algorithm we give here a short description of its basic ideas.

5.3.1 The Algorithm of Earley

In principle the algorithm uses a breadth first strategy, in contrast to the depth first strategy used for example by the recursive descent method (Davie and Morrison, 1982). This means that all possible derivations starting with the start symbol are analysed and tested against the input. Whenever a derivation ends because it doesn't match the input, it is simply left out and no backtracking need be done. Furthermore the algorithm notices if in two distinct derivations the same nonterminal has to be parsed at the same input position and avoids repetition of the analysis. By parsing identical parts in different derivations only once, the algorithm can provide good time bounds.

In (Earley, 1970) the algorithm is described with a look-ahead. Since we process highly dynamic grammars this feature is of no practical use for us.

The basic principle of the algorithm is to represent each started derivation in a *state* and to process these states into following states. A state is given by a production, a position in that production that shows how much of the production has already been parsed and a token position in the input where the parsing of the production started.

For example:

A : 'x' .B   [0]

is the informal representation of a state, in which the position is given by the dot and the start position by the number in brackets. This example can be read as 'the attempt to find nonterminal *A* at token position 0 (at the beginning of the input) has lead to the state where an *x* has been found in the input and where the nonterminal *B* has to be found next'. If a production of *B* matches the next tokens after the *x* the nonterminal *A* is found at the beginning of the input stream.

All states with the same current token position (not the same start position) are collected in a *state set* . In the previous example one token is scanned (the *x*), so the current token position is '1' and the state would be in state set S_1. The states in a state set are processed in sequence, adding states to the current or to the next state set. Depending on the position of the dot in a state, one of three operations is called to process that state.

The *predictor* is called when the dot is in front of a nonterminal. For every alternative of this nonterminal a new state is added to the current state set, all with the dot at the beginning of the alternative. The *scanner* is called whenever a dot occurs in front of a terminal symbol in a production. The scanner tries to match the next token in the input stream against this terminal and adds a new state to the next state set, if it is successful. Whenever the dot occurs at the end of a production this means that it is fully parsed and the nonterminal on the left side has been found. In this case the *completer* is called to add all those states of a previous state set with the dot before the nonterminal to the current state set with the dot now behind it.

5.3.2 Our Extensions

Distributed Scanners

In order to support distributed scanners we had to extend the information held in a state. The token position is implicitly given by the state set number in which the state is held but with distributed scanners it is possible that in two distinct derivations the n-th token starts at different character positions in the input stream. We illustrate this with the following example:

Assume the rules:

(i) COMMAND : 'lsw' PARAM
(ii) COMMAND : 'ls' /id(directory)/

are part of the current grammar. The user types '`lsw myfile`' to print 'myfile' on a laserwriter. Rule (i) matches, so for this rule 'lsw' is the first and 'myfile' is the second token. But if there is a directory named 'w', rule (ii) can also be derived. For this rule the command is split into three tokens. The second token is 'w' and it starts at character position 3 and ends at position 4.

The current character position of a state and the character position at which the current production started must therefore be added to a state and be handled by the predictor, scanner and completer.

Transformations

The algorithm as described in (Earley, 1970) is only a recogniser. Further extensions had to be made to make it a transformation algorithm. One possibility was to keep all produced parse trees. After the parsing is done, a selection can be made as to which parse tree is the right one (if there is more than one) and with that parse tree the transformation could be done.

In our implementation the transformation is done as soon as a production has been completed, but instead of doing string manipulations we represent the transformation in a tree structure.

Only after a successful parse of the whole command line the costly expansion of this transformation tree into the resulting string carried out.

With the special kind of grammar we are processing in MASK (dynamic and distributed) in combination with transformations, situations can occur where the time bound can no longer be guaranteed to be O(n^3). This can be shown by a simple example:

```
A : A A
  | 'a' --> '0'
  | 'a' --> '1' .
```

The grammar part of this rules defines all strings of length > 0 consisting only of a's. But for a given string of length n there are 2^n transformation possibilities. The string 'aaa' for example can be transformed into `000,001,010 ... 111`.

This means that the time bound of the algorithm with transformations has in theory become exponential. We do not believe this to be a serious problem in practice because situations like this with two recursive rules with identical productions and different transformations are most unlikely to occur in a normal command language environment.

Optimisation

In the first state set (S_0) are those states which have not scanned a terminal. Therefore this state set does not depend on the input and need not to be constructed again for each command line (only on a change of the grammar). This is interesting especially in our command language environment where there are often many different possibilities to start derivations, but after the first token is scanned most of them can be excluded. In our test environment S_0 was about 5 or 6 times larger than the following state sets. This means, if we assume an average number of tokens of 5 per command, we achieve a time saving of about 50%.

We implemented this optimisation by processing S_0 only on a change of the grammar. While processing this state set the scanner calls are deferred. In this way parsing of a command line can start directly with the first scanner call in S_0.

5.4 Ambiguities

Our strategy for the resolution of ambiguities is based on a simple, but effective assumption and is only practical with distributed scanners: the less tokens a derivation needs to match the command line the more specialised it is and thus should be selected. A simple example:

```
COMMAND : 'list' /id/ --> '#2.list'.

COMMAND : 'list vars' --> 'var_manager.list'.
```

If both rules are active and a user enters 'list vars', the second rule matches the command line in state set S_1, the first one in state set S_2. Because the second rule is obviously more specialised it should be chosen.

For the transformation algorithm this means that as soon as a derivation has matched the start symbol against the whole command line the parsing process can stop. Only the current state set must be finished. If there is more than one finished derivation in the last state set an interactive resolution is adopted. We believe that no automatic mechanism will provide an acceptable success rate.

To demonstrate how an interactive resolution is performed we assume that the two syntax rules:

```
COMMAND : 'list' --> 'var_manager.list'.
COMMAND : 'list' --> 'directory.list'.
```

are both part of the current grammar. If a user enters the command 'list' he or she is prompted with the following output:

```
$ list
Warning: ambiguous command.
<1> var_manager.list
<2> directory.list
```

By entering the appropriate number the user can select the correct transformation.

5.5 Error Handling

The current version of the MASK system offers no mechanism for the rule writer to provide explicit error messages with the rules. Instead an implicit mechanism tries to give informative error messages based on various assumptions:

- If at least one literal has been scanned in a derivation, the next expected symbol for all derivations in the last state set is reported.
- If no literal has been scanned the parsing of the command line repeated without type checking. If this time the parsing succeeds a type error is reported.
- If no literal is scanned and no type error can be reported only a general error message can be given. If the command line is partly parsed the message is:

  ```
  ERROR: illegal Symbol for this position.
  ```

  with a marker at the command line position where the parsing stopped. If no token at all could be parsed the only message the error mechanism can provide is:

  ```
  ERROR: illegal command.
  ```

5.6 MASK in use

To demonstrate the utility of the system we give some examples of commands now in regular use on the MONADS-PC. Some users are totally unaware that the command syntax is distributed in various class definitions rather than defined centrally in the CLI. Others have themselves defined special syntax rules in their own class definitions.

Examples:

```
date --> date.show
```

This command displays the current date and time by calling the object 'date'.

```
mkdir dir --> directory_manager.create  dir
```

A new directory with the name 'dir' is created.

```
var i:integer --> var_manager.define_var "i" "integer"
```

This command defines a new CLI variable 'i' of type 'integer'.

```
i:=j+1 --> var_manager.assign_integer "i" (int_lib.add j 1)
```

The Pascal-like syntax of this command   transformed into the appropriate calls for manipulating integer values and variables.

```
x? --> help_manager.object_help "x"
```

This concise syntax can be used to obtain help about an object.

Of course the advantages of such a flexible command syntax are of no use if the users reject the system as having an unacceptable response time. In fact the MASK system was installed on the MONADS-PC without informing the users (this was possible because syntax rules exist which allow the previously accepted CLI syntax to be used as before) and no user complained of a noticeable increase in the system's response time.

Command lines tend to be very short on average so that the time required for parsing and transformation depends primarily on the number of rules in the current grammar. Tests have shown that the time for transformation (with the rules currently installed on the system) is roughly proportional to the number of active rules, ie. to the number of open objects. This explains the good response times since in general relatively few objects are simultaneously open.

Even if many objects are kept open, however, the advantages of the flexible syntax outweigh the disadvantages. With over 100 active grammar rules the time required for parsing and transformation has been measured at 1.1 seconds on the MONADS-PC (a 0.5 MIPS machine). This may seem a long time but considering the time required to enter:

```
var_manager.assign_integer "i" (int_lib.add j 1)
```

in comparison with:

```
i:=j+1
```

the overall gain is nevertheless clear. (Not to mention the time required to consider what exactly must be entered in the former case.)

The time for the transformation is partly accounted for by the fact that the MONADS-PC hardware is over 10 years old. On modern hardware a time of some 40 milliseconds for the same number of rules could be expected. This would then hardly be noticeable.

6. Conclusion and Future Work

An object invocation language based on the concept of object-oriented language definition as described in this paper can be extended flexibly to adapt to new user domains and individual user preferences. Not only the set of available commands, but also the notation for the invocation of the commands can be tailored to a particular user's needs at a particular time. The notation to be used in invoking objects of a certain class is described together with the rest of the class definition. The notations understood by various classes are combined with each other to define the command language.

The implementation described in this paper demonstrates that this approach is not only useful in theory but is also realisable with sufficient efficiency to be accepted by normal users. It has also shown that normal users are capable of making use of the system to define appropriate commands for their own needs.

Further work must be done on handling errors in input lines. The error messages produced currently although formally correct, are not particularly helpful. The rule definitions could be extended to incorporate the texts of error messages which should be output if the parsing fails at a certain point.

Another interesting area of further research is the possibility of transforming not only a user's input to the system but also the system's output to the user, whether this be results returned by an operation on an object, or exceptions caused by inappropriate parameters. This would conform to the aim of unifying command languages and *response languages* into a single dialogue language as proposed in (Mac an Airchinnigh, 1986) et al.

References

ANSI (1983) "The Programming Language Ada - Reference Manual", American National Standards Institut, Inc., ANSI/MIL-STD 1815A, 1983, LNCS, 155, Springer-Verlag

Balzert, H. (1988) "Introduction to Software Ergonomics", De Gruyter, Berlin.

Beetem, A. & Beetem, J. (1989) "Introduction to the Galaxy Language", IEEE Software, May 1989, pp. 55-62.

Cardelli, L., Matthes, F. and Abadi, M. (1993) "Extensible Grammars for Language Specialization. in Database Programming Languages", Database Programming Languages Workshop, New York City: Springer-Verlag.

Davie, T. & Morrison, R. (1982) "Recursive Descent Compiling", Ellis Horwood Limited, Chichester, England.

Earley, J. (1970) "An Efficient Context-Free Parsing Algorithm", Communications of the ACM, 13, 2, pp. 94-102.

Evered, M.(1993) "An Object-Oriented Approach to Language Definition", Technical Report, University of Bremen.

Goldberg, A. & Robson, D. (1989) "Smalltalk-80: the language", Addison-Wesley Series in Computer Science.

Hitchens, M. (1989) "The Structure of a Command Language Interpreter", Proc. of the 4th IFIP Working Conf. on User Interfaces, North Holland.

IBM. (1966) "IBM System/360 Operating System: job control language", Form C28-6539.

Keedy, J.L. & Thomson, J.V. (1986) "Command Interpretation and Invocation in an Information-Hiding System", Foundation for Human Computer Communication, Elsevier Science Publishers B.,V. (North Holland). IFIP 1986, pp 278-292.

Leavenworth, B.M. (1966) "Syntax Macros and Extended Translation", CACM, **9**(11): pp. 790-793.

Mac an Airchinnigh, M. (1986) "Responses - the Weak Part of Dialogues?", in Hopper, K. & Newman, I.A. (eds), Foundation for Human-Computer Communication, North-Holland, Amsterdam.

Ritchie, D. M. & Thomson, K. (1974) "The UNIX Timesharing System", Communications of the ACM, 17, 7, pp. 365-375.

Rosenberg, J. & Abramson, D. A. (1985) "MONADS-PC: A Capability Based Workstation to Support Software Engineering", Proceedings of the 18th Hawaii International Conference on Systems Sciences, 515-522.

Standish, T.A. (1975) "Extensibility in Language Design", ACM Sigplan Notices, July 1975, pp. 18-21.

Stroustrup, B. (1989) "The C++ Programming Language", Addison-Wesley, Bonn.

Appendix A: An Example of a Class Definition

```
var_manager = class
  function define_var (varname  : string;
                       basetype : string;
                       logtype  : string) : integer;
  function delete_var (varname : string) : boolean;
  procedure list (long : boolean);
  function get_type (    varname  : string;
                     var basetype : string;
                     var logtype  : string) : integer;
  function assign_int (varname : string;
                       val     : integer) : integer;
    ...

{syntax
  COMMAND : /id/ ':' logical_type basetype --> '#name.define_var "#1" "#4" "#3"'
          | 'vl' mode --> '#name.list #2'
          | 'typeof' /id/ --> '#name.get_type "#2" s1 s2; ?s1; ?s2'
          | 'vrm' vrm_params --> #2
          | assignment.
  basetype : 'integer' | 'boolean' | 'real' | 'modcap' | 'string'.
  logical_type : /id/ '=' --> '#1' | .
  mode : 'long' --> 'true' | 'short' --> 'false'
       |  --> 'true'.
  vrm_params : /id/ --> '#name.delete_var "#1"'
             | vrm_params /id/ --> '#1; #name.delete_var "#2"'.
  assignment : /id(string)/ ':=' STRING --> '#name.assign_string "#1" #3'
             | /id(integer)/ ':=' INTEGER --> '#name.assign_int "#1" #3'
             | /id(modcap)/ ':=' MODCAP --> '#name.assign_modcap "#1" #3'
             | /id(boolean)/ ':=' BOOLEAN --> '#name.assign_bool "#1" #3'.
}
end; { class var_manager  }
```

Appendix B: The Syntax of the MASK Meta-Language

```
syntax_part      ::=  'syntax' { syntax_rule } .
syntax_rule      ::=  head ':' alternatives '.' .
head             ::=  private_nonterm | public_nonterm .
alternatives     ::=  alternative {'|' alternative } .
alternative      ::=  production [ '-->' transformation ] .
production       ::=  { element } .
element          ::=  literal | nonterm | ident_scan | external_scan .
literal          ::=  ' ' ' { character } ' ' '  .
nonterm          ::=  private_nonterm | public_nonterm .
private_nonterm  ::=  lowercase_ident .
public_nonterm   ::=  uppercase_ident .
ident_scan       ::=  '/id'  [ '(' identifier ')' ] '/' .
external_scan    ::=  '/' identifier '.' identifier '/' .
transformation   ::=  transf_string { transf_string } .
transf_string    ::=  ' ' ' { transf_char | special_char | reference } ' ' '  .
transf_char      ::=  character except ('#', '/', '{'}') .
special_char     ::=  '/' character .
reference        ::=  '#' digit | '#name' .
identifier       ::=  letter { letter | digit | '_' } .
uppercase_ident  ::=  uppercase_char { uppercase_char | digit | '_' } .
lowercase_ident  ::=  lowercase_char { lowercase_char | digit | '_' } .
letter           ::=  uppercase_char | lowercase_char .
uppercase_char   ::=  'A' .. 'Z' .
lowercase_char   ::=  'a' .. 'z' .
digit            ::=  '0' .. '9' .
character        ::=  (any ASCII character) .
```