SYNTAX MACROS FOR EFFICIENT IMPLEMENTATION OF SPECIAL PURPOSE PROGRAMMING LANGUAGES

by

E. Kantorowitz and Tamar Netzer

Technical Report No. 62

December 1975

This work was supported by U.S. Army grant DA-ERO-591-74-G0043.
Techniques for low cost implementation of special purpose languages are proposed and demonstrated. The implemented languages are characterized by high compilation speeds and reasonable error handling. Efficient object code may be achieved.

A new language is implemented by adding "few" rules to the grammar of an existing core language. These rules are specified in a simple way through macros using the same meta language, e.g. BNF, and symbols as employed for the core language. Coding and debugging of these macros is facilitated by a high level "macro time language". Restrictions on macro calls counteract unclear programming style and enable fast compilations. Typechecking of macro parameters enables a simple and efficient error analysis. Efficiency is further achieved through the integration of the macro processor in the compiler.

In a system implemented for PL360 the macro processor was built in after the scanner such that it uses the syntactic entities recognized by the scanner as its atoms. The macro processor added only 10 K Bytes object code to the 42 K Byte compiler. Observed compilation speeds of implemented language were high, i.e. more than half the speeds of corresponding PL360 programs.
1. INTRODUCTION

The costs of solving problems in computers are considerably reduced when employing programming languages enabling solutions to be expressed in a clear and "natural" way. Problems of extreme complexity may probably only be solved when an appropriate language is employed.

An expensive method for fitting a programming language to the problems encountered may therefore be useful and it may enable a promising problem solving approach demonstrated in (4) where structured programming is combined with the implementation of a language specially suited for expressing the solution.

The most promising approach for inexpensive language implementation is based on language extension, since the implementor has only to add "few" facilities to an already implemented core language, and because the extensions may be specified in a simple way by utilizing the existing terminology of the core language. The topic of this paper is extension techniques that enable implementation of languages with "modest" programmer efforts, and such that the implemented languages are characterized by "high" compilation speeds and when needed also by efficient object code.

The principles of the method described in this work were developed by E. Kantorowitz while the implementation of the practical system to be discussed is due to T. Netzer (7).

Some scholars make a distinction between "definitional facilities", whose purpose is to improve the syntax of a language and "extension
facilities", which enable computations that are not possible in the core language (3). We shall, however, employ the term "extension" for both cases. The major bottleneck in problem solving is probably not in the computational capabilities of current languages, but rather in limitations in our mental abilities, which may be enhanced by introducing appropriate syntax and concepts in the employed languages.

The value of language extension was realized early and current languages provide subroutine or macro facilities for this purpose. When a programmer defines a macro or a subroutine the programming language is enriched with the calls to this new macro or subroutine respectively.

The usual syntax of macro and subroutine calls, i.e. a name followed by a list of parameters, is however, not always satisfactory. To illustrate this let us consider a call of a macro for a user defined type of addition.

```
Add A, B, C
```

It is not obvious whether this call means \( A := B + C \) or \( C := A + B \). The same unclarity occurs when the extension is realized by a subroutine, e.g.

```
CALL ADD (A,B,C);
```

This unsatisfactory syntax may be avoided in ALGOL 68 by defining an addition operator denoted by \(+\), whereafter additions are expressed in a clear way, e.g. \( A := B + C \). In the case of macros the unsatisfactory macro call syntax may be avoided by employing "template macros", such as in ML/; (1), where the delimiters between the parameters in the macro
call are not mere commas, but meaningful symbols, e.g.

\[
\text{LET } A := B + C;
\]

In this example, the four character strings, "LET", "="", "+" and ";" are the delimiters, which are specified in the macro definition. The parameters are the character strings that are found between the delimiters, i.e. "A", "B" and "C" in the example. The parameters are regarded by these macro processors as mere strings, and their properties are neither checked nor utilized.

A further step was the introduction of syntax macros by Cheatham (2) and Leavenworth (5). In syntax macros, the "syntactic type" of every parameter is specified, e.g.

\[
\text{\langle call of macro LET\rangle ::= LET \langle variable\rangle ::= \langle variable\rangle + \langle variable\rangle;}
\]

where L, E, T, :=, + and ;

are terminal symbols of the core language to be extended, while the three non-terminal symbols \langle variable\rangle represent three parameters. It is assumed that the non-terminal symbol \langle variable\rangle is defined in the core language, such that the processor may check whether the actual parameters in calls of the above macro are legal variable identifiers.

A further advantage is that the above declaration of the macro head is at the same time a rule in BNF notation for the construct added to the language, i.e. a language is extended by adding a macro defined rule to the set of
rules by which the core language is defined. If a notation other than BNF is employed for the definition of the grammar of the core language, this other notation should obviously be used for the macro definitions.

Implementation of new grammar rules by syntax macro enables syntax extensions more general than possible with operator definitions in ALGOL 68 and procedure definitions in ALGOL 60 like languages. Macros further enable the selection of the most appropriate object code at macro time such that considerable savings at run time are achieved. On the other hand, manipulation with grammar rules through syntax macros requires more competence than required for defining operators in ALGOL 68 and procedures in ALGOL 60 like languages. Programmers with the required knowledge are however provided by universities and other institutions in considerable numbers.

The use of macros for extension of programming language was the topic of a previous study (4), where a general purpose macro processor ML/l was employed in a quite successful experimental system. One of the conclusions was that it would be of great advantage to employ a macro processor fitted to the purpose. The required advances were a high level macro time language that facilitates coding and debugging of language extensions as well as significant improvements in compilation speed and error recovery capabilities of the implemented language. A syntax macro processor that meets these requirements was developed for the PL/360 language (9) and is discussed in the next section.
2. **PL360 SYNTAX MACROS**

The reason for choosing a machine oriented system implementation language as our core language is that it may be extended into anything between a sophisticated specialist tool to an English like language for non-programmers. PL360 was chosen because it is regarded to be useful (8) and is at the same time formally defined (9).

The structure of macro definition is

```
<definition initiator>
<macro head>
mas
<macro body>
mend
```

, where

```
<definition initiator> ::= mtypedef|mdef
```

The two `<definition initiator>s` correspond to the two kinds of macro definitions provided, i.e. `mtypedef` for introducing data types not available in PL360, and `mdef` for defining new types of statements. Note that key words used for macro definitions start with \texttt{m}, e.g. `mas` is a "macro as".

Let us now as an example introduce the data type \texttt{complex} not available in PL360 by the definition:
If we hereafter declare

```c
complex A, B, C;
```

an array of two reals is allocated to each of the three variables, and their names, i.e. A, B and C are entered in the symbol table with the tag "complex", such that their types may later be looked up.

The above definition of the data type complex contains a "formal macro parameter" [A]. The "type" of a formal macro parameter is specified in its first appearance in the head of the macro definition. The type of the parameter [A] in the above definition of complex is thus iden (identifier). A detailed discussion of the type concept is given in the subsequent section.

We see that macro definitions are written in an extended BNF notation. One of the added notations is that of a "parameter vector"

```c
<separator> <parameter> ¶
```

The above definition of the data type complex thus contains the "formal vector" ¶, [A] ¶. The corresponding "actual parameter vector" in the example declaration shown above is

A, B, C
Let us now assume that complex numbers have been introduced by the above definition, and that we wish to define a new statement type, *addcomp*, for adding two complex numbers.

```plaintext
mdef addcomp [A complex] := [B complex] + [C complex];
```

```plaintext
mas

F0:=B(0) + C(0); [A](0):=F0;
F0:=B(4) + C(4); [A](4):=F0;
```

```plaintext
mend
```

where *F0* is an IBM/360 register for performing real arithmetic. The two components of the complex numbers [A] [B] and [C] have the indices 0 and 4 in the machine oriented PL/360 language because a real occupies 4 bytes. If we, hereafter, write the statement

```plaintext
addcomp X:Y+Z;
```

The macro processor will substitute it by the code

```plaintext
F0:=Y(0)+Z(0); X(0):=F0;
F0:=Y(4)+Z(4); X(4):=F0;
```

and check that the actual parameters *X*, *Y* and *Z* have been declared as complex.

If we want to define a new statement type, *LETcomp*, that operates on any number of complex numbers the "repetition parenthesis", { }, which is another BNF extension, is used.
A statement of this type may be

\[ \text{LETcomp } P := Q + R + S - T; \]

which the macro processor will substitute by the PL360 statements:

\[
\begin{align*}
F_0 & := Q(0) + R(0) + S(0) - T(0); & P(0) & := F_0; \\
F_0 & := Q(4) + R(4) + S(4) - T(0); & P(4) & := F_0;
\end{align*}
\]

If it is required to give the operator \( \ast \) a higher priority than + and - some extra code in the "macro time language", which will be discussed later, must be written.

3. TYPES OF PARAMETERS

The type of a parameter is the set of symbols from which the actual parameter must be chosen. A formal parameter may thus be regarded as a non terminal symbol added to PL360 while the type specify the terminal and non terminal that may substitute it. In the implemented system such a set of symbols, i.e. the type, is specified as:
1. A set of PL360 terminal symbols, e.g. \([\text{OP} + | - | \times]\). An actual parameter that corresponds to the formal parameter \(\text{OP}\) must be one of the three symbols +, - and \(\times\).

2. One of the following non terminal symbols of PL360: identifiers, numbers, strings, shift operators, arithmetic operators, relation operators, e.g. \([Q \text{idem}]\). The actual parameter must be an identifier.

3. One of the following PL360 non terminals each of which represent a subset of the type identifier: short integer, integer, real, long real, character, integer register, real register, long real register, macro, procedure, function, e.g. \([A \text{real} | \text{Int}]\). The actual parameter must be the identifier of a variable declared as either a real or an integer.

The processor checks that the type of corresponding formal and actual parameters match. A "tight typechecking" may be avoided by specifying the type by an appropriate large set of symbols.

4. **THE MACRO TIME LANGUAGE (MTL)**

Concepts: The body of a macro is composed of some strings and of instructions in a "Macro Time Language" (MTL) that operates on the strings, whereby the macro expansion that substitutes the macro call is produced. The MTL is thus a string manipulation language and macro definitions may be regarded as the procedures of the MTL.
In a previous study (4), where a general purpose macro processor was employed for language extensions, the writing and debugging of the MTL proved to be quite expensive. It was therefore decided to design a MTL that is high level for our purpose and at the same time as simple as possible.

The implemented MTL has integer variables and expressions, conditionals and loops. Macro definitions play the role of procedures. Conditionals (IF-THEN-ELSE-FI constructs) are used for specifying code that will only be produced under certain conditions. Let us as an example modify the previously discussed macro addcomp for adding complex numbers such that it will also operate correctly on reals and such that the result is in any case a complex number.

```
mdef addcomp [A complex] := [B real|complex] + [C real complex];
mas
    FO := [B](0) + [C](0); A](0) := FO;
    FO := mif type ([B]=complex)  mthen [B](4) melse 0 mfi
         + mif type ([C]=complex)  mthen [C](4) melse 0 mfi;
    mend
```

The key work mfi is used in a similar way to fi in ALGOL 68 to avoid the problem of dangling eles.

If we call

```
addcomp [U := V+W;
```

where V is real and W a complex then the code
FO := V (0) + W (0); \quad V (0) := FO;
FO := 0 + W (4); \quad V (4) := FO;

will result. In the third of the produced statements a zero is added to FO for no purpose. Such a waste may be avoided by defining the macro such that the object code that is most efficient for the actual case is produced. Such a code selection involves, however, some macro time processing. The macro definer may thus choose between emphasizing production of efficient object code or achieving high compilation speeds. A macro time loop is implemented by a `mfor` construct which is similar to the `for` construct of ALGOL.

5. Structure and Restrictions of Macro Calls

With some macro processors any encountered symbol may initiate a new macro call, and a macro call may be generated through the expansion of other calls. This enables a powerful but tricky coding, and implies high processing costs because every symbol involves a look up in the macro table. The need for such a tricky coding for our purposes is questionable and it was therefore decided to avoid the high bug probability and high processing costs involved. A number of restrictions were, therefore, imposed:

1. A macro call may not contain other calls. A macro call thus represent a statement explicitly, i.e. a programmer that looks at a program does not have to speculate on how the statements may look after
possible macro expansions.

This restriction involves no significant limitations on power of expression and computation because names and parameters of macros to be called during the macro expansion may be parameters in the macro calls. Recursive macro calls are also permitted.

2. A macro call represents a complete statement. Macros representing parts of statements were proposed by (2) and (5), but were not implemented for PL360 because they were not regarded to be of basic importance and because there are some possibilities for misuse. This decision should probably be reconsidered in future systems.

3. A macro call is initiated by its name and terminated by a ; . The syntax of macro calls is thus

<macro name> { <parameter> | <terminal symbol of PL360> } ;

A similar restriction is found in languages like BASIC and COURSEWRITER, where every statement starts with a keyword. The restriction could quite easily be removed for the case of assignment statements by permitting them to start with the identifiers of any of the declared variables. The case of assignment statements could be generalized by permitting macro calls to start with a parameter, whose type is the set of the possible initialization symbols. Such a generalization is, however, not recommendable because it may quite easily be misused such that ambiguous languages result.
5. **IMPLEMENTATION METHOD**

One of the keys to the efficiency of the system is the savings achieved by integration of the macro processor into the compiler. As illustrated by Fig. 1, the macro processor is built in between the scanner (the lexical analyzer) and the parser (the syntactical analyzer) of the existing PL360 compiler (6). The macro processor operates therefore on the PL360 symbols that have already been recognized by the scanner. Compared to systems with a macro pre-processor we save a separate scanner for the macro pre-processor and obtain at the same time a "syntax macro processor" that operates on the syntactic entities recognized by the scanner. Furthermore, the macro processor utilizes the symbol tables produced through the compilation to check the types of variables. This is possible because when a variable is encountered its description is already in the symbol table. The reasons are the requirements of PL360 to declare a variable before its usage and the one pass compiler that thus process the declaration before the usage of the corresponding variable.

Typechecking of variables may probably also be implemented in the more difficult cases where multi pass compilers are employed or when declarations are not required by the language. The compiler must obviously also in these cases be organized such that variables are entered into the symbol tables before the processing of the macro calls that contain them.
Transformation of the program through the processors of the compiler. The three processors are shown as rectangular frames.

Source program as string of characters

The scanner (Lexical analyzer)

Program as a string of numerical codes, which represent PL360 symbols recognized by the scanner (identifiers, numbers etc.) as well as macro definitions and calls

The macro processor

Program as a string of numerical codes, which represent PL360 symbols of the kind recognized by the scanner. (The macro processor has removed the macro definitions and substituted the macro calls by the corresponding macro expansions).

The parser + semantic analyzer + code generator

Program in object code
7. PARSING, ERROR DETECTION AND RECOVERY

A macro call is composed of a macro name followed by a sequence of actual parameters, whose order and type is specified in the head of the macro definition. The parsing is therefore initiated by the macro name and the identification of the actual parameters is achieved in a single and fast scan through the call (a top down technique). If a symbol that does not fit to the macro definition is encountered three error hypotheses are checked: That the encountered symbol is missing, that it is mutated, and that there is a superfluous symbol. The analysis is based on two assumptions:

1. There is at the most one error at the actual program point.

2. Every two adjacent formal parameters in the "neighbourhood" of the error are of different types.

The correct error hypothesis is decided by a simple algorithm based on one look ahead. See Fig. 2.
Error Analysis

<table>
<thead>
<tr>
<th>Position in the string</th>
<th>i-1</th>
<th>i</th>
<th>i+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of formal parameter</td>
<td>$T_{i-1}$</td>
<td>$T_i$</td>
<td>$T_{i+1}$</td>
</tr>
<tr>
<td><strong>Error cases:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. $A_i$ is a mutation</td>
<td>$A_{i-1}$</td>
<td>$A_{i}+\text{MUT}$</td>
<td>$A_{i+1}$</td>
</tr>
<tr>
<td>2. $A_i$ missing, $A_{i+1}$ is in its place</td>
<td>$A_{i-1}$</td>
<td>$A_{i}+A_{i+1}$</td>
<td>$A_{i+1}+A_{i+2}$</td>
</tr>
<tr>
<td>3. Superfluous symbol in place of $A_i$</td>
<td>$A_{i-1}$</td>
<td>$A_{i}+\text{SUP}$</td>
<td>$A_{i+1}+A_{i}$</td>
</tr>
<tr>
<td>$A_i$ pushed to the place of $A_{i+1}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Error Analysis Algorithm:

```python
if type($A_i$) = $T_{i+1}$ then <$A_i$ is missing> else
if type($A_{i+1}$) = $T_i$ then <$A_i$ is superfluous> else
if type($A_{i-1}$) = $T_{i-1}$ and type($A_{i+1}$) = $T_{i+1}$ then <mutation> else
<more than one error>;
```

Since the two assumptions made are probably met in most cases the simple algorithm of Fig. 2 was employed. A more complex algorithm that analyzes the errors also in the case of "two errors at the same program point" was not considered justifiable in the experimental system.
Error correction was implemented only in the case where it may be done accurately, i.e. when the erroneous symbol is a superfluous one it is discarded, and when the error is in a character specified explicitly in the macro definition, e.g. if the + is an addcomp (see Section 2) is mutated a + is inserted instead of the wrong symbol. In other cases dummies of correct type are inserted such that compilation may be continued and more errors detected in the same run (error recovery).

8. CONTRACTIBILITY AND SAVING OF MACRO DEFINITIONS

Two useful facilities, which were not implemented are devices for language contraction and saving of macro definitions.

Contraction is the ability to implement a language by specifying a subset of an existing one (4). Let us as an example assume that PL360 is extended by an English like language for non programmers. The resulting language still contains PL360, and errors which happen to be legal PL360 constructs will not be detected by the compiler. It is therefore recommended to remove unneeded language elements by contraction. The contraction techniques discussed in (4) are not fitted for our system. Other solutions based on modifications of the macro processor may probably be devised.

It is an obvious waste to process the macro definitions that define a language for every program compiled. It is therefore recommended to implement a facility for saving sets of processed macro definitions.
9. COSTS OF THE EXTENSION MECHANISM

The macro processor added 10 K bytes of object code to the original compiler, which comprises of about 42 K bytes of code and 20 K bytes of tables.

In a number of "worst case analyses" the compilation time of a program composed solely of macro definitions and calls was compared to that of another version of the same program where macros were not employed at all. The cases were, however, constructed such that the macro expansion did not involve calls of further macros. In these "worst cases" the definition and expansion of macros added about 100% to the compilation time. Considering the speed of the original PL360 compiler (it compiles itself in 3 seconds on the IBM/370-168) we may conclude that fast compilation time may be expected for languages implemented by macro extensions of PL360.

EXPERIENCE WITH THE SYSTEM

The system was used by undergraduate students of the Computer Science Department of Technion - Israel Institute of Technology for the implementation of a subset of the COURSEWRITER language, which is a widespread language for Computer Assisted Instruction. The effort required for the implementation was quite small which is due to

1. The students had a prior knowledge of the PL360 language and of the BNF
notation. They had therefore only to learn the few BNF extensions and the simple high level macro time language.

2. The concept of language extension by syntax macros was easily understood and used.

3. Macro definitions in the employed notation proved to give a clear picture of the implemented language extensions.

4. The language was implemented in a number of not too difficult steps.

In each of these implementation steps a more abstract level of the language is defined by means of a more primitive one. The implementation of COURSE-WRITER with these "linguistic structured programming techniques" is described in some detail in (4), where, however, a general purpose macro processor was employed for language extensions.

The use of PL360 syntax macros showed in this application a number of advantages over the general purpose macro processor employed in (4). The high level macro time language and the extended BNF notation thus facilitated the coding of the macros. It is further expected that the restrictions on macro calls and typechecking will result in considerably higher compilation speeds and in a more efficient error handling.

**SUMMARY AND CONCLUSION**

The principal contribution of this work is the development of techniques for low cost implementation of special purpose languages. The implemented languages
are, furthermore, characterized by high compilation speeds and reasonable error handling. Efficient object code may be produced when needed. These low cost techniques for language implementation enable a problem solving methodology based on the development of a language specially fitted for expressing the solution ("Linguistic Structured Programming" described in (4)).

The results are due to

1. The implementation of a new language involves only the addition of relatively few features to an already implemented core language.

2. The implementor has only to learn to use few new facilities. The extensions are thus defined by means of the same meta language (BNF in the case of PL360) and the same symbols as employed for the definition of the core language.

3. The extension by syntax macros is a conceptually simple procedure, which was easily understood and used. The macro head defines the syntax of the new non terminal symbol added to the language, while the macro body defines its semantics using the terms of the core language.

4. A high level macro time language facilitates coding and debugging of the language extensions.

5. Specification of the types of the macro parameters enables a parsing algorithm that utilizes the syntactic properties of the actual parameter. The type specifications further enable extensive error checking including correctness of types of program variables.
6. A simple and quite efficient error analysis algorithm is based on checking the hypothesis of mutated, missing and superfluous symbol. It is furthermore based on the assumption of "maximum one error per program point" and "adjacent symbols in a program have different types". A reasonable error recovery is expected in most cases and an accurate error correction in some other cases.

7. Restrictions on macro calls facilitates fast compilation and counteracts a programming style that is difficult to analyze. A statement implemented by a macro call is thus the statement that the programmer writes and looks at and not the result of later macro expansions. The restrictions therefore probably reduce both error probability and the mean costs for debugging an error. The ability to express solutions through macro calls is probably only insignificantly reduced by these restrictions:

8. A compiler organization where the three "compiler processors" the scanner, the macro processor and the parser utilize the services of each other. This enables the extension mechanism to be implemented with little code and such that fast compilation of the programs of the implemented language result. In the actual system the macro processor is built in after the scanner enabling the macro processor to use the syntactic entities recognized by the scanner as its atoms. The macro processor further utilizes the information stored in the symbol tables by the parser. Further savings could have been achieved if the same parsing algorithm was adopted for both the macro processor and the parser.

The developed syntax macro processor contrasts with common general purpose
macro processors in a number of ways. Restrictions on macro calling, specification of types of macro parameters and a high level macro time language have thus been introduced. As a result a macro processor specially fitted for translation and debugging of programs has been achieved. By integrating such a macro processor into a compiler an efficient implementation of a language of two grammar levels resulted. Student experiments showed that the macro level of the grammar facilitates adaptations of the language to actual needs and is useful for solving non trivial programming problems as well as for implementation of special purpose languages.
REFERENCES


(4) KANTOROWITZ, E., GONEN, A., SAGI, Y., Extensible and Contractible Programming Languages.

(5) LEAVENWORTH, B.M., Syntax Macros and Extended Translation, Comm., ACM, 9, (November 1966)

(6) MALCOLM, M.A., PL360 a Programming Language for the IBM/360, Stanford University, May 1971.

(7) NETZER, T., Development of a Macro Processor for the PL360 Compiler (in Hebrew), M.Sc. Thesis, Computer Science Dept., Technion - Israel Institute of Technology, Haifa, Israel
