Abstraction Mechanisms Supported by a Macro Processor

YOSHIAKI FUKAZAWA*

A macro language and its processor are described for various kinds of abstraction techniques.

It is very attractive to support the use of abstraction in program construction by means of a general-purpose macro processor with special facilities. For that purpose, a macro language MACLAM (A Macro Language for Abstraction Mechanisms) has been designed and implemented. Attention of this macro language is focused on data as well as on control.

MACLAM supports three kinds of abstraction techniques: (1) procedural abstraction, (2) data abstraction, for which a user can define new data types, define associated operations and protect them from illegal operations, and (3) syntactic abstraction to develop powerful mode of expressions and to give a method for sequencing arbitrary actions. MACLAM offers a procedure and some special functions for these abstraction techniques.

The purpose of this paper is to illustrate the utilities of abstractions supported by MACLAM and to provide an informal introduction to MACLAM through some examples.

1. Introduction

In current language design research, data abstraction languages and very high level languages are major concerns of many researchers [24]. The emphasis in both types of languages is increasing the extent to which a programmer may state "what" is to be done rather than the details of "how" it is to be done. The very high level languages (VHLL) are an attempt to substantially raise the level at which a programmer states a problem solution. The data abstraction languages have been designed to support a methodology [6, 23], in which programs are developed by means of problem decomposition based on the recognition of abstractions. The concept of abstract data types was first implemented as a class in the language Simula 67 [5]. Other implementations of this basic concept have been included in many recently proposed languages; for example, clusters in the language CLU [13], forms in the language Alphard [17], classes in Concurrent Pascal [8], and so on [10, 14, 22].

In order to add data abstraction facilities to an existing programming language, some attempts have already been made. Practically, either of the following approaches is adopted: implementing a special purpose pre-processor [4, 25] or modifying the compiler currently used [1, 26].

On the other hand, various kinds of macro languages have been developed since the middle of the 60s [3]. The advent of high level languages has not removed the need for macro facilities. Actually, there are a number of macro based extensibility schemes for high level languages [18].

Some macro languages, in which complicated macro call statements are available, have been developed [2, 16, 19, 21]. Above all, syntax-macros by Leavenworth [12] are especially notable, since they allow a programmer to extend the syntax and semantics of the given base language by new statements or expressions.

It is particularly appealing for us to connect merits of some kinds of abstraction techniques with those of a macro processor. For this purpose, a macro language MACLAM (a MACro Language for Abstraction Mechanisms) has been designed. MACLAM is a general-purpose and syntax-directed macro language, and attention of this language is focused on data as well as on control. The MACLAM processor is implemented as a complete pre-processor for various kinds of base language processors.

In the next section, we explain the goals of MACLAM and then outline necessary facilities for them. Subsequently, an informal introduction to MACLAM and the current implementation of the MACLAM system are described. We conclude by discussing the quality and the efficiency of abstractions supported by macro facilities of MACLAM.

2. Goals and Framework of MACLAM

MACLAM is a macro language intended to support three kinds of abstractions: data abstractions, syntactic abstractions and procedural abstractions.

To support data abstractions, the following three facilities are essential:

(a) the facility to define new data types in terms of primitive data types of the base language or other user-defined data types,

(b) the facility to define operations associated with these new data types by primitive operations, and

(c) the facility to protect defined data type objects from illegal operations.

As regards abstract data definitions ((a) and (b) above), in MACLAM, definitions of abstract data types,

*Department of Information Science, Sagami Institute of Technology, Tsujido Nishikaigan 1-1-25, Fujisawa, 251, Japan.

definitions of operations on these new data types, and actual operations for the variables of the created abstract data types must be all expressed as macro call statements. Therefore, in developing programs by means of a programming language without data abstraction facilities, first of all, a proficient programmer called a system programmer must define a macro, which is called a data abstraction definition macro from its nature. In this macro definition, the system programmer must incorporate such facilities as to generate macro definitions for translating abstract data types into specified primitive data types, to generate macro definitions for translating operations for abstract data objects into suitable subroutine call or function call statements, and to modify definitions of abstract data operations into legal subroutine definitions or function definitions of the base language. Moreover, it specifies a syntax for the definitions of abstract data types and their operations. In Section 3.2, necessary facilities for the data abstraction definition macro and its example are given in detail.

As concerns data protection, some symbol-table-handling facilities are necessary by all means for a complete pre-processor like the MACLAM processor. Therefore some special macro-time functions are available in MACLAM. These functions are usually used in data abstraction definition macros. So it is enough for only the system programmer to be in charge of protection of abstract data. Detailed mechanisms for protection will be described in Section 3.4.

That is to say, data abstraction facilities are integrated into the base language through the data abstraction definition macro. At the same time, it enables ordinary programmers to define their own data types and operations in suitable form.

As the result, the data abstraction definition macro becomes relatively complicated. According to our experiences, it is very desirable to be able to specify abstract data definitions in a similar fashion to the notation of the base language. Therefore, it is usually sufficient to define only one data abstraction definition macro for each base language. The complexity can be nothing serious.

The objective of syntactic abstractions is to develop a powerful mode of expressions and to give a method for sequencing arbitrary actions. For example, to provide a CLU-like iterator or a REPEAT-UNTIL statement for PL/1 (F-compiler) is syntactic abstraction.

A syntax-directed macro processor provides syntactic abstraction facilities for a base language. A complete syntax-directed parser is, however, time-consuming even if an effective parser is developed. So in MACLAM, the first symbol of a macro call statement must be a basic symbol called a prefix macro delimiter.

For syntactic abstractions, it is necessary to specify a syntax of the base language in advance. The syntax notation employed here follows a BNF-like notation.

Macro languages originally aim at providing open procedures. In addition, conventional languages support the procedural abstraction well, through procedures or subroutines. Therefore MACLAM does not have a special facility for procedural abstractions except for macro expansions.

3. Specifications for MACLAM

The purpose of this paper is to describe the utilities of the three kinds of abstractions supported by MACLAM in program construction, and to provide an informal introduction to MACLAM. We do not attempt a complete description of the language; rather we concentrate on constructs that support the abstractions. For details, the reader is referred to the MACLAM reference manual [15]. This manual is independent of a base language, because of the language independency of the MACLAM processor.

The syntax of MACLAM has been influenced by that of PL/1 and Pascal. To illustrate the following examples, PL/1 is selected as the base language.

3.1 Defining and Calling Macros

An example of a macro definition is shown in Fig. 1(a). It defines the REPEAT···UNTIL··· control structure using the IF···THEN···ELSE···statement and the GO TO statements. Namely it means one of the syntactic abstractions.

Character strings between reserved words "%DEF" and "%BODY" are called a macro templete. "%BODY" starts a macro body and it continues until "%DEFEND" appears.

The macro templete specifies the syntactic type and the proto-type structure of the macro definition. Most of the syntactic types are ⟨STATEMENT⟩, which is the default type. The proto-type structure must begin with a prefix macro delimiter. In this example, "REPEAT" is the prefix macro delimiter. It may be a little restrictive. But at present almost all statements of conventional languages are assumed to commence with fixed key-words, for example "DO", "IF" and "READ". In some languages (PL/1, Fortran, etc.), these key-words are not reserved words, and semantics is determined by the content. In MACLAM, this property is reserved, because it is a syntax-directed macro language.

The character % starts one parameter. The parameter

```
%DEF <STATEMENT>
     REPEAT <:ST: %S-STATEMENT :> UNTIL %E-EXPRESSION ;
%BODY   LABEL1%INDEX :
            <:ST: %S :>
            IF %E
                THEN GO TO LABEL2%INDEX ;
                ELSE GO TO LABEL1%INDEX ;
        LABEL2%INDEX :
%DEFEND
```

(a)

```
REPEAT  NEXT = NEXT + 1 ;
        MAX = MAX - 1 ;
        UNTIL NEXT > MAX ;
```

(b)

Fig. 1 Example of a MACLAM program. (a) A macro definition. (b) A macro call statement for Fig. 1(a).

consists of its name and attribute. The definer of the macro can access the parameter by means of the name in the macro body. The attribute must be previously specified in a BNF-like notation or the syntactic type of the other macro templete.

Braces, which are expressed by character pairs <: and :> in EBCDIC, denote an occurrence one or more times in succession. Each successive occurrence in the macro templete is associated with a corresponding successive occurrence in the macro body by a unique name (a character string without a colon). This name appears just after an open brace (or <:) and delimited by a colon. If the correspondence is unambiguous, the name may be omitted.

Other control structures in macro templetes are square blackets which denote optional occurrences and vertical strokes which separate alternatives. Character pairs (: and :) are used as synonyms for [and] respectively. Concerning optional occurrences, correspondences between the macro templete and the macro body are similarly specified to successive occurrences.

In order to generate a unique symbol, the macro-time special function %INDEX is attached to the end of a symbol in a macro body. The facility of %INDEX is analogous to that of &SYSNDX in IBM OS/370 assembly language [11].

Another example in Fig. 2 is a part of a macro definition. In MACLAM, macro-time variables preceded by % must be declared explicitly in the %DECLARE statements. The %IF statement, the %THEN statement and so on are available macro-time control statements. A pair of double quotation marks is used to suppress the macro facilities.

This macro definition is called the data abstraction definition macro according to the usage. The facilities of the data abstraction definition macro and the usage of the built-in functions %OPERATION, %OPERAND and %VARIABLE in Fig. 2 are given in Section 3.2 and 3.4, respectively.

```
%DEF ABSTRACT DATA DEFINITION ;
     TYPE %TYPE-IDENTIFIER = %STRUCTURE-IDENTIFIER %ATTRIBUTE ;
   <:1: %LABEL-IDENTIFIER : PROC (<:1:( %PARM1-IDENTIFIER
     <:2: , %PARM2-IDENTIFIER :1> ) :) (<:2: %RETURNATTRIBUTE :) ;
       <:3: %STATEMENT-STATEMENT :>
        END  %LABEL2-IDENTIFIER ; :>
   END ABSTRACT DATA DEFINITION ;
%BODY
  "%DEF  DECLARE  %NAME-IDENTIFIER "  %TYPE :
  "%BODY"
  %IF %STRUCTURE = 'ARRAY'
      %THEN  "DECLARE %NAME " %ATTRIBUTE ;
             "%VARIABLE ( %NAME , " %TYPE )
      %ELSE  %IF %STRUCTURE = 'STRUCTURE'
                 %THEN "DECLARE 1 %NAME " %ATTRIBUTE ;
                       "%VARIABLE ( %NAME , " %TYPE )
                 %ELSE "DECLARE %NAME " %ATTRIBUTE
                       "%VARIABLE ( %NAME , " %TYPE )
             %END
  %END                                                          (A)
  "%DEFEND"
  %DECLARE %PARMCTR INTEGER INITIAL (1) ;
  <:1: %OPERATION ( %TYPE,%LABEL,%PARMCTR,%PARM1 )          ) (C)
    %PARMCTR = %PARMCTR + 1 ;
    %IF %PARM1 ¬= NULL
        %THEN "%DEF"  DECLARE %PARM1 "%PARMTYPE"
              "%BODY" DECLARE %PARM1 "%PARMTYPE"
                  "%OPERAND ( " %LABEL,%PARM1 ",%PARMTYPE )"  ) (D)
              "%DEFEND"
  %END                                                          (B)
  <:2: %OPERATION(%TYPE,%LABEL,%PARMCTR,%PARM2)             ) (C')
       %PARMCTR = %PARMCTR + 1 ;
             "%DEF"  DECLARE %PARM2 "%PARMTYPE"
             "%BODY" DECLARE %PARM2 "%PARMTYPE"
                 "%OPERAND ( " %LABEL,%PARM2 ",%PARMTYPE )"  ) (D')
             "%DEFEND"
  :>
```

Fig. 2 Data abstraction definition macro for a subset of PL/1 (part).

3.2 Data Abstraction Definition Macros

Data abstraction facilities are integrated into a base language through a data abstraction definition macro. A system programmer defines the data abstraction definition macro and shows ordinary programmers how to specify new data types and relevant operations. Fig. 2 illustrates a part of its simple example for the subset of PL/1.

It is necessary for the system programmer to define the data abstraction definition macro so as to incorporate the following facilities.

(1) To generate a macro definition which translates a defined abstract data type into specified built-in data types. See Part A in Fig. 2. This generated macro definition is named the declaration macro. It is called and expanded when declarations of abstract data type variables appear.

(2) To transform definitions of operations for abstract data into corresponding subroutine definitions or function definitions of the base language.

(3) To generate a macro definition which translates operations for abstract data types into suitable subroutine call statements or function call statements. This generated macro definition, which is named the operation macro, is called at operations for abstract data.

(4) To protect abstract data from illegal operations (see Part B in Fig. 2). Detailed facilities of MACLAM for data protection will be explained in Section 3.4.

The relation of the data abstraction definition macro and generated macro definitions is shown in Fig. 3. In Fig. 3, subscripts (1), (2) and (3) at arrows indicate the above description. This relation is explained with program examples in Section 3.3. Figure numbers in Fig. 3 correspond to those examples.

Since specifications for data abstractions are determined by defining the data abstraction definition macro,

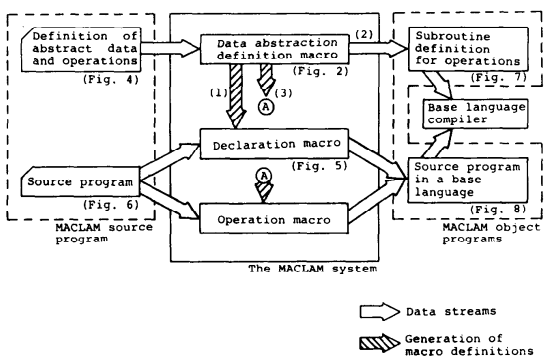

Fig. 3 Translation process of an abstract data definition and a source program by the data abstraction definition macro.

various kinds of data abstraction techniques can be made available. For example, a system programmer can ever provide mechanisms allowing user-defined abstract data types to be parameterized, if necessary.

Concerning data abstractions, some error detection facilities can be incorporated in a data abstraction definition macro by a system programmer. For example, when global data types of the base language (EXTERNAL in PL/1, COMMON in Fortran, and so on) are specified in abstract data type definitions, warning messages can be issued.

3.3 Definition of Abstract Data and Its Usage

A user of MACLAM may define abstract data types which conform to a data abstraction definition macro. One example is illustrated in Fig. 4. It defines data type "SET" and relevant operations. The "SET" type data object is a set of bit strings, each with an associated element. The operations are "FULL", "DELETE", "IN" and "PRINT". For example, "DELETE" is used to delete the first parameter from the set indicated by the second parameter. In general, a definer of abstract data can provide as many operations as seems reasonable.

If abstract data type "SET" is defined as in Fig. 4, the declaration macro in Fig. 5 is generated by the data abstraction definition macro in Fig. 2. The declaration macro aims at replacing abstract data type "SET" with built-in data type array and bit strings. The operation macro, which translates operations on abstract data types into subroutine call or function call statements, is as simple of this declaration macro.

An example shown in Fig. 6 is a program to compute prime numbers based on the algorithm known as Eratosthenes' sieve. The method of Eratosthenes is first to put all numbers in the sieve and repeat the followings until the sieve is empty: Select and remove the smallest

```
ABSTRACT DATA DEFINITION ;
        TYPE  SET = ARRAY ( 0:1000 ) BIT ( 1 ) ;
    FULL : PROC ( SIEVE ) ;
           DCL  SIEVE     SET ;
           DCL  COUNTER   BIN FIXED ( 15,0 ) ;
         DO COUNTER = 0 TO 1000 ;
             SIEVE ( COUNTER ) = '1'B ;
         END ;
    END FULL ;
    DELETE : PROC ( NUMBER , SIEVE ) ;
             DCL  SIEVE      SET ;
             DCL  NUMBER     BIN FIXED ( 15,0 ) ;
           SIEVE ( NUMBER ) = '0'B ;
    END DELETE ;
    IN : PROC ( NUMBER , SIEVE ) RETURNS ( BIT ( 1 ) ) ;
         DCL  SIEVE     SET ;
         DCL  NUMBER    BIN FIXED ( 15,0 ) ;
      IF SIEVE ( NUMBER ) = '1'B
          THEN  RETURN ( '1'B ) ;
          ELSE  RETURN ( '0'B ) ;
    END IN ;
    PRINT : PROC ( SIEVE ) ;
            DCL  SIEVE     SET ;
            DCL  COUNTER   BIN FIXED ( 15,0 ) ;
         DO COUNTER = 0 TO 1000 ;
             IF SIEVE ( COUNTER ) = '1'B
                 THEN PUT LIST ( COUNTER ) ;
         END ;
      END PRINT ;
  END ABSTRACT DATA DEFINITION ;
```

Fig. 4 Example of an abstract data definition based on Fig. 2.

```
%DEF  DECLARE  %NAME-IDENTIFIER  SET ;
%BODY  DECLARE  %NAME  ( 0:1000 ) BIT ( 1 ) ;
%DEFEND
```

Fig. 5 Declaration macro generated by the data abstraction definition macro in Fig. 2.

```
PRIME : PROC OPTIONS ( MAIN ) ;
        DCL  TARGET   BIN FIXED ( 15,0 ) ;
        DCL  NEXT     BIN FIXED ( 15,0 ) INIT ( 2 ) ;
        DCL  SIEVE    SET ;
     FULL ( SIEVE ) ;
     DELETE ( 0 , SIEVE ) ;
     DELETE ( 1 , SIEVE ) ;
     DO WHILE ( NEXT <= SQRT (1000) ) ;
          DO TARGET = 2*NEXT TO 1000 BY NEXT ;
               DELETE ( TARGET , SIEVE ) ;
          END ;
          REPEAT NEXT = NEXT + 1 ;
                 UNTIL IN ( NEXT , SIEVE ) ;
     END ;
     PRINT ( SIEVE ) ;
END PRIME ;
```

Fig. 6 Sample program for the usage of abstract data type variables.

```
FULL : PROC ( SIEVE ) ;
       DCL  SIEVE     ( 0:1000 ) BIT ( 1 ) ;
       DCL  COUNTER   BIN FIXED ( 15,0 ) ;
    DO COUNTER = 0 TO 1000 ;
        SIEVE ( COUNTER ) = '1'B ;
    END ;
END FULL ;
DELETE : PROC ( NUMBER , SIEVE ) ;
         DCL  SIEVE      ( 0:1000 ) BIT ( 1 ) ;
         DCL  NUMBER     BIN FIXED ( 15,0 ) ;
      SIEVE ( NUMBER ) = '0'B ;
END DELETE ;
IN : PROC ( NUMBER , SIEVE ) RETURNS ( BIT ( 1 ) ) ;
     DCL  SIEVE     ( 0:1000 ) BIT ( 1 ) ;
     DCL  NUMBER    BIN FIXED ( 15,0 ) ;
  IF SIEVE ( NUMBER ) = '1'B
      THEN RETURN ( '1'B ) ;
      ELSE RETURN ( '0'B ) ;
END IN ;
PRINT : PROC ( SIEVE ) ;
        DCL  SIEVE     ( 0:1000 ) BIT ( 1 ) ;
        DCL  COUNTER   BIN FIXED ( 15,0 ) ;
     DO COUNTER = 0 TO 1000 ;
         IF SIEVE ( COUNTER ) = '1'B
             THEN PUT LIST ( COUNTER ) ;
     END ;
END PRINT ;
```

Fig. 7 MACLAM object program for Fig. 4.

```
PRIME : PROC OPTIONS ( MAIN ) ;
        DCL  TARGET   BIN FIXED ( 15,0 ) ;
        DCL  NEXT     BIN FIXED ( 15,0 ) INIT ( 2 ) ;
        DCL  SIEVE    ( 0:1000 ) BIT ( 1 ) ;
        DCL  IN       RETURNS ( BIT ( 1 ) ) ;
     CALL FULL ( SIEVE ) ;
     CALL DELETE ( 0 , SIEVE ) ;
     CALL DELETE ( 1 , SIEVE ) ;
     DO WHILE ( NEXT <= SQRT (1000) ) ;
          DO TARGET = 2*NEXT TO 1000 BY NEXT ;
               CALL DELETE ( TARGET , SIEVE ) ;
          END ;
          LABEL101 :
               NEXT = NEXT + 1 ;
               IF IN ( NEXT , SIEVE )
                     THEN GO TO LABEL201 ;
                     ELSE GO TO LABEL101 ;
          LABEL201 :
     END ;
     CALL PRINT ( SIEVE ) ;
END PRIME ;
```

Fig. 8 MACLAM object program for Fig. 6.

number remaining in the sieve, and then step through the sieve, removing all multiples of that number. This program was first used by Hoare [9] to illustrate data structuring. It contains the "SET" type data defined in Fig. 4 and the REPEAT···UNTIL···statement in Fig. 1.

The abstract data definition in Fig. 4 is translated into Fig. 7 by the data abstraction definition macro in Fig. 2. The program shown in Fig. 6 is transformed into Fig. 8 by the declaration macro (Fig. 5) and the operation macros, which are generated by the data abstraction definition macro, and the REPEAT···UNTIL···macro in Fig. 1(a). Fig. 7 and Fig. 8 are ordinary PL/1 programs, which are compiled by a PL/1 compiler.

3.4 Protection of Abstract Data

In order to protect abstract data from violations, the MACLAM processor supports two protection tables

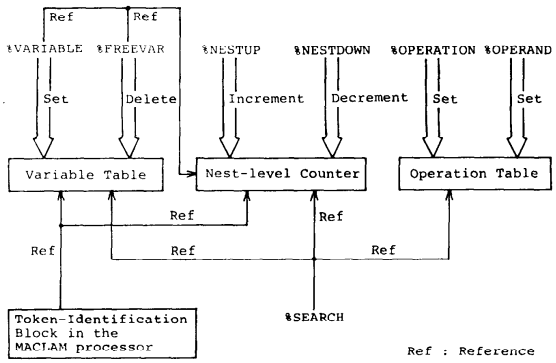

Fig. 9 Relation of the protection tables and the relavant operations.

(the Variable Table and the Operation Table) and five protection functions (%OPERATION, %OPERAND, %VARIABLE, %FREEVAR and %SEARCH). These are abstract data types and their operations prepared for system programmers, provided by the MACLAM system. The relation of these tables and functions is illustrated in Fig. 9.

Illegal operations to abstract data are divided into two classes: violations caused by operations for other kinds of abstract data and by built-in operations of the base language.

In order that abstract data may be protected from violations caused by other abstract data type operations, the following protection mechanisms are applied by a system programmer.

The protection tables consist of the Variable Table and the Operation Table. The former is used to protect abstract data from illegal operations. By means of the macro-time special function %VARIABLE, a variable declared as an abstract data type must be set in it. The latter is used to examine operations for abstract data whether they have suitable types of variables as operands. %OPERATION and %OPERAND are prepared for registering the operation name and expected types of operands into this table. In a data abstraction definition macro, a system programmer must check the type of the operands using the %SEARCH function.

A part of this protection mechanism is illustrated in Part B of Fig. 2. When the data abstraction definition macro is called, namely when an abstract data type and related operations are defined, the data type name, the operation name, the parameter names and so on are set in the Operation Table by the protection function %OPERATION in Part C and C' of Fig. 2. Successively, in Part D and D', a macro definition is generated. This macro definition is called when the parameter of the operation is declared, and sets the operation name, the parameter name and its data type name in the Operation Table by means of the protection function %OPERAND. Protection of abstract data from illegal operations, which are defined for other abstract data type, is performed after preparation like this.

In the MACLAM system, all operations to variables whose attributes are abstract data types are expected to be macro call statements. As regards built-in operations, type checking is performed by utilizing this property. Namely, every token (a lexical entity) fetched by the MACLAM processor is examined whether it is declared as a defined data type in the Variable Table. This part of the MACLAM processor is called the Token-Identification Block. Moreover, a system programmer must define a data abstraction definition macro, in which expanded text is rescanned just after operands of the operation for abstract data. Because of the above-mentioned mechanism, if a token which is registered in the Variable Table is fetched by the Token-Identification Block, the MACLAM system can recognize a protection error caused by a built-in operation.

In addition to these, the Nest-level Counter and two macro-time functions %NESTUP and %NESTDOWN are provided for visibility of variables. This facility is very important when the MACLAM processor is applied to a base language which supports block structures. The protection function %FREEVAR is used in order to delete all variables defined at the current nest level in the Variable Table. In a base language without special scope rules, this facility is unnecessary.

In a number of languages, data protection is not perfectly performed for some kinds of data types. An example is where a PL/1 pointer, upon which a data structure is based, is assigned an address of a different data structure. Applying the MACLAM system can make perfect protection possible in such a case. In the above case, a user can define a new pointer type which consists of a pointer and its tag, and check the tag in an operation of the new pointer type variable for protection.

4. Structure of the MACLAM System

Fig. 10 is a general flow of the MACLAM system. The inputs of this system are classified into three categories: base language definitions, macro definitions and a source program including various kinds of abstractions. The outputs of the system are the MACLAM object programs, which are compiled by a base language processor, and three output lists such as a source list and a cross reference list.

According to their usage, macro definitions are classified into permanent macro definitions (PMD) and temporary macro definitions (TMD). Permanent macro definitions consist of data abstraction definition macros, and other system macros if necessary. On the other hand, temporary macro definitions are called only in the source program currently processed.

For effective processing, definitions of a base language are classified into two levels: the lexical information (LIBL) and the syntactic information (SIBL). Specifications of the syntactic information of the base language are based on the BNF notation.

An ordinary programmer has only to define temporary

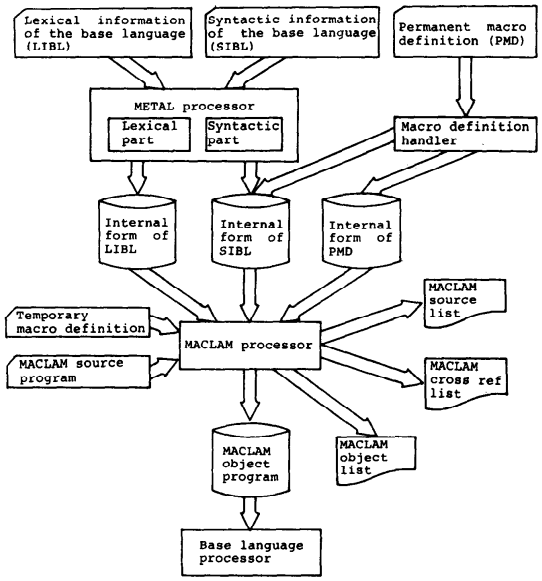

Fig. 10 Structure of the MACLAM system.

macros and to write a source program. Other definitions (i.e. base language definitions and permanent macro definitions) are all performed by a system programmer previously.

The lexical information (the syntactic information) is translated into the internal form of the lexical information (the syntactic information) in the lexical part (the syntactic part, respectively) of the Meta Language Processor, which is called the METAL processor. Experiences have shown, however, that people do not often write correct grammars. Much of the METAL processor is devoted to the production of rather complete diagnostic messages for the various error conditions.

The MACLAM processor parses given source programs and macro definitions (both permanent macros and user-defined temporary macros) into tokens under the internal form of the lexical information of the base language. Input texts broken down into tokens are syntax-directedly checked whether a defined macro is matched.

Efficiency is the most important problem for a processor which acts as a complete pre-processor to a compiler and utilizes the syntactic information of the base language. It results from a syntactic analysis which is performed twice: on the macro processor and the base language compiler. Besides, the more flexible syntax of macro call statements becomes available, the more a complicated recognition algorithm becomes necessary.

As the results of above trade-off, the notation of macro call statements described in the previous section was fixed and the following syntactic analysis algorithm is employed. It is founded on Earley's algorithm [7] and further improved by utilizing the restriction of macro call statements: prefix macro delimiters and lexical information of the base language.

The MACLAM processor has been implemented on the IBM 4341 processor system in Waseda University. It is written in PL/1 for portability.

5. Experience and Discussion

In view of our experiences, various results were obtained.

In the beginning, we selected PL/1 as the base language. When the MACLAM system is applied to some base language, a system programmer must define the syntax of the base language and a suitable data abstraction definition macro in advance. The syntax definition of PL/1 is so complicated [20] that we adopted a subset. SIBL for the subset is composed of about 280 lines. Current version of the data abstraction definition macro for this subset of PL/1 is only about 70 lines, though it is complicated. It took about 0.5 man month to define and test the macro. The MACLAM system spends approximately 39 ms and 31 ms on the sample programs in Fig. 4 and Fig. 6, respectively. Note that our PL/1 compiler processes the programs of Fig. 7 and Fig. 8 in 13 ms and 18 ms, respectively.

Our second target is to apply the MACLAM system to Pascal. The syntax of Pascal is much simpler than that of PL/1. SIBL for Pascal is composed of about 90 lines. With an effort of only 3 man day, we have been able to define the data abstraction definition macro for Pascal, whose size is comparable to that for PL/1. The ease is due to the facts that the syntax of data abstraction for Pascal is essentially the same as that for PL/1, and that the data abstraction definition macro can be defined independently of the base language compiler. We believe that the more data abstraction definition macros we define for various languages, the less effort we have to make.

A Pascal program, which is equivalent to the program in Fig. 6, is processed in 20 ms by the MACLAM processor. Therefore, it turned out that the greater part of the processing time is spent for the syntax-directed processing.

We can not compare these results with those of data abstraction languages, because we do not have a processor of a data abstraction language presently. But the approach, which is to attach data abstraction facilities to the existing language, is very useful until data abstraction languages like Ada are arailable. We also believe that it is easier to define a data abstraction definition macro than to develop a special purpose pre-processor or to modify a compiler currently used. But our scheme is maybe less satisfactory in efficiency than the others.

The experiment has also pointed up some of the important issues to pursue in developing this scheme further. These are categorized as follows: results from inherent properties of macro languages, and limitations in abstract data definitions.

MACLAM was not able to get over a number of in-

herent defects of macro languages of this kind. Some of them are described below.

(1) A macro call statement requires a prefix macro delimiter in MACLAM. Therefore an infix operator is not permitted on the variables of defined data types.

(2) The MACLAM processor can never detect collisions of names of identifiers (variables, labels and so on) in two texts expanded by the MACLAM processor or in a source program and in an expanded text. Macro-time special function %INDEX is prepared for this purpose. Ultimately we can find no alternate way but to define macros carefully.

(3) Statement numbers contained in error messages detected by a base language compiler do not correspond to source statement numbers. A cross reference list indicating these correspondences is given by the MACLAM processor, if necessary.

(4) The property of MACLAM is restricted by that of a base language. For example, MACLAM can not support a separate compilation facility for a language without this facility.

Other weaknesses of MACLAM are common characteristics to all data abstraction languages. For example, defined data types must be expressed by a combination of built-in data types of the base language.

6. Conclusion

In this paper, we have described some facilities available in the MACLAM system for the use of three kinds of abstractions.

Our first object was to apply the MACLAM system to PL/1, since it lacks data abstraction facilities though it is powerful. As a result, we got a very powerful programming language. A specification language and a hardware description language have been implemented in this language. Though Pascal has some data structuring facilities, stronger data abstraction facilities are obtained by the application of the MACLAM system.

Further, we want to apply MACLAM to Fortran or Cobol. Consequently, it will be necessary to modify the MACLAM processor for the purpose of appending a field specification facility.

Compiler writing systems are deeply related to our approach. Surely, simple compilers can be generated by making a practical application of this system. But the generated compiler would not only be less efficient but would also have a few shortcomings in code generation and optimization. We emphasize that many advantages are in extending a base language, for example, to add a data abstraction facility by a macro processor with special facilities.

While a new language with strong abstraction facilities is very useful, we believe that it is more practical to provide them for a familiar language.

Acknowledgments

The author wishes to thank A. Nojima and R. Yamagata for their contributions to designing and developing the MACLAM system, and is very greatful to K. Utsunomiya of Tsukuba University, M. Sakakura of Waseda University and S. Iwata of Tokai University for their helpful advices. He is also indebted to Professor T. Kadokura for his encouragement. Finally, the comments of the referees were most helpful.

References

1. Banatre, M. et al. An Experience in Implementing Abstract Data types, *Softw. Pract. Exper.*, 11, 315–320 (1981).
2. Brown, P. J. The ML/1 Macro Processor, *Comm. ACM.* 10, 10, 618–623 (1967).
3. Brown, P. J. Macro Processors and Techniques for Portable Software, John Wiley & Sons (1974).
4. Burton, W. A FORTRAN Preprocessor to Support Encapsulated Data Abstraction Definitions, *Comput. J.*, 22, 4, 307–312 (1979).
5. Dahl, O.-J. Hierarchical Program Structures, in Structured Programming, Academic Press, New York (1972).
6. Dijkstra, E. W. Notes on Structured Programming, ibid.
7. Earley, J. An Efficient Context-free Parsing Algorithm, *Comm. ACM*, 13, 2, 94–102 (1970).
8. Hansen, P. B. The Programming Language Concurrent Pascal, *IEEE Trans. on Soft. Eng.*, SE-1, 2, 199–207 (1975).
9. Hoare, C. A. R. Notes on Data Structuring, ibid.
10. Ichbiah, J. D. et al. Reference Manual for the Ada Programming Language, SIGPLAN Notices, 14, 6 (June 1979).
11. IBM OS/VS-DOS/VS-VM/370 Assembler Language, File No. S370–21, GC33–4010–4 (Jan. 1975).
12. Leavenworth, B. M. Syntax Macros and Extended Translation, *Comm. ACM*, 9, 11, 790–793 (1966).
13. Liskov, B. et al. Abstraction Mechanisms in CLU, *Comm. ACM*, 20, 8, 564–576 (1977).
14. Musser, D. R. Abstract Data Type Specification in the AFFIRM System, *IEEE Trans. on Soft. Eng.*, SE-6, 1, 24–31 (1980).
15. Nojima, A. and Fukazawa, Y. MACLAM Reference Manual (in Japanese), Dept. of Electrical Engineering, Waseca University (1981).
16. Sassa, M. A Pattern Matching Macro Processor, *Softw. Pract. Exper.*, 9, 439–456 (1979).
17. Shaw, M. Wulf, W. A. and London, R. L. Abstraction and Verification in Alphard, Defining and Specifying Iteration and Generators, *Comm. ACM*, 20, 8, 553–564 (1977).
18. Solntseff, N. and Yezerski, A. A Survey of Extensible Programming Languages, *Ann.* Rev. Auto. Program., 7, 5, 267–307 (1974).
19. Strachey, C. A General Purpose Macrogenerator, *Comput. J.*, 8, 3, 225–241 (1965).
20. Urschler, G. Concrete Syntax of PL/1, *IBM Technical Report* TR25.096 (1969).
21. Waite, W. M. The Mobile Programming System: STAGE2, *Comm. ACM*, 13, 7, 415–421 (1970).
22. Wegbreit, B. The Treatment of Data Types in EL1, *Comm. ACM*, 17, 5, 251–264 (1974).
23. Wirth, N. Program Development by Stepwise Refinement, *Comm. ACM*, 14, 4, 221–227 (1971).
24. Wulf, W. A. Trends in the Design and Implementation of Programming Languages, *IEEE Computer*, 13, 1, 14–24 (1980).
25. Young, S. J. Improving the Structure of Large Pascal Programs, *Softw. Pract. Exper.*, 11, 913–927 (1981).
26. Zelkowitz, M. V. and Larsen, H. J. Implementation of a Capability-Based Data Abstraction, *IEEE Trans. on Soft. Eng.*, SE-4, 1, 56–64 (1978).

(Received December 23, 1981: revised December 1, 1982)