Computer Languages Vol. 6, pp. 139 to 153, 1981
Printed in Great Britain.

0096-0551/81/030139-15$02.00/0

IMPLEMENTATION OF LANGUAGE ENHANCEMENTS*

MARVIN V. ZELKOWITZ[1,2] and JAMES R. LYLE[1]

[1]Department of Computer Science, University of Maryland, College Park, MD 20742 and
[2]Institute for Computer Sciences and Technology National Bureau of Standards, Washington, DC, U.S.A.

(*Received* 15 *April* 1981; *revision received* 17 *June* 1981)

Abstract—Data abstractions have been proposed as a means to enhance program modularity. The implementation of such new features to an existing language is typically handled by either rewriting large portions of an existing compiler or by using a preprocessor to translate the extensions into the standard language. The first technique is expensive to implement while the latter is usually slow and clumsy to use. In this paper a data abstraction addition to PL/1 is described and a hybrid implementation is given. A minimal set of primitive features are added to the compiler and the other extensions are added via an internal macro processor that expands the new syntax into the existing language.

Compiler design Data abstraction Language extensions Specifications

1. INTRODUCTION

PROGRAMMING language design is by no means a dead issue. As people use various compilers, new and different features are continually desired. This enhancement process, usually miscalled maintenance, often costs more than the original development of the product itself. Thus most compilers continue to evolve as their user communities expand and as they are used on new and different applications.

In addition, knowledge about programming is changing. Such topics as structured programming, encapsulated data types, and program verification are all recent developments. While these are often incorporated as features in new languages and compilers, there is also a desire to add related features into existing compilers and languages so that programmers are able to use them on current projects.

This paper describes a data abstraction addition to PL/1 that adds the protection of abstract data types yet still retains the basic PL/1 programming style. The implementation of these extensions as well as some results using them in a student environment is given.

The problem, therefore, is how to extend current compilers to accept these new techniques. The usual approach towards data abstraction has been to design and build a new language having these ideas embedded in the semantics of the language [1]. There is no way to truly use abstract data types in the more commonly used languages. FORTRAN has no user types at all and PL/1 has a structure declaration that is only an approximation of a type. While Pascal has a type declaration, it permits any user of the type to also access the components of variables of that type.

The major problem with designing new languages is that the resulting languages are not FORTRAN, COBOL, or to a limited extent, PL/1 or Pascal. Thus most of these languages have limited visibility and usefulness outside of their development environments. The situation with the new Department of Defense language Ada may be different; however, the complexity of the language means that the first quality compilers are several years away.

An alternative approach can be achieved by the use of preprocessors to extend existing languages. A language like Pascal or PL/1 as the desired primitives, and good programs

* Research supported in part by Air Force Office of Scientific Research grant F49620-80-C-001 to the University of Maryland. Computer time provided in part by the Computer Science Center of the University of Maryland.

can be written in these languages if they are used correctly, much like good structured programs can be written, although with some difficulty, in FORTRAN. Thus one can design an extension to PL/1 or Pascal which enforces good data abstractions and then use a preprocessor to translate the program into proper PL/1 or Pascal for compilation and execution. The PL/1 extensions, described in Refs [2, 3], use this approach.

There are two problems with this last process. One is that the compiler is always processing a program different from the program written by the programmer. In most cases, error messages, diagnostics and output will be oriented towards the preprocessor output that was compiled and not the programmer's source program input. This usually means that the programmers must often carry two program listings—input and output from the preprocessor. Given an error condition during execution, the programmer finds the place of the error on the preprocessor output listing, and then reverse translates it into the original source input language—a most cumbersome process. While it is possible to alter the compiler somewhat to account for this, the procedure is not easy since the compiler has no knowledge about the preprocessor input language.

A second drawback to this approach is the overhead and complexity involved. The preprocessor must read, parse and process the entire program and output a new program, which again gets read, parsed and processed by the compiler—a slow and redundant step. The user must invoke several system commands to accomplish this. This added cost and complexity often gets in the way of its use. While the extensions described in [2] were efficiently implemented, the syntax depended very heavily on PL/1 pointer variables and so the full protection against improper usage of abstract variables was not present.

A variant of this process is used, however, with many data management systems. A data base language processor often converts data base queries into another language (often COBOL) and then compiles that translated language without the user realizing that a translation has taken place. For example, the INGRES relational data base system [4] translates SEQUEL queries into the language C as subroutine calls to the INGRES system routines and then compiles the C program. Thus while a preprocessor is being used, its application is as a translator to convert a different query language into an existing compilable language.

This second macro-processing approach does have merit if the two drawbacks mentioned above can be eliminated, and this paper describes one such implementation. A macro-processor, unknown to the user, was embedded within a PL/1 compiler to translate new statements into standard PL/1 text. Since the compiler did the translation, the programmer only had one source listing to contend with. The compiler knew about the original source program so it could key messages to the appropriate source line, and since the macros were stored in parsed form by the compiler, the source program was processed only once—leading to a very efficient system when compared to the usual macro-processing systems. A final attribute of the system was that the substitutions were into PL/1, thus only the first parsing phase of the compiler was altered. The semantics or code generation phases only processed "standard" PL/1 and were mostly untouched. Thus the extensions that will be described in this paper were added in minimal time when compared to the cost of writing a new compiler.

In Section 2 the extensions to PL/1 that we desired to investigate are described. The problems in implementing these extensions led to the compiler modifications that are described in Section 3. Section 4 describes the implementation of these extensions in one particular PL/1 compiler.

2. EXTENSIONS TO PL/1

In order to improve on the process of writing good programs, there is great interest in designing new languages that enforce desired restrictions on the programming process to achieve a better product (e.g. Alphard [5]; Euclid [6]; Gypsy [7]; Ada [8]; Clu [9]).

One of the major developments in this area is the encapsulated data type, commonly called a data abstraction. Rather than viewing a program as a collection of statements, a

program is now viewed as a set of data types with operations defined using these objects of these types as arguments. The only thing one can do with such an object of an abstract type is to pass it as an argument to one of the operations defined for the type, much as using a primitive operation on a primitive type (like INTEGER). Only these operations know the details of the type: it is not possible to look at the internal representation much like it is not possible in standard FORTRAN to look at the internal bits of an integer. Therefore, control over accessing the internal structure of the type is much more localized, much like structured programming limits the control flow. This enhances such desired characteristics as information hiding and modularity [10].

Our goal was to design and implement a practical data abstraction language as extensions to the PLUM PL 1 compiler at the University of Maryland [11]. These extensions are of two types: (1) Abstract data types: and (2) Specifications.

2.1 *Abstract data types*

A program consists of a set of abstractions—structured objects—and a set of operations on these objects. An abstraction has the syntax:

```
⟨name⟩: ABSTRACTION:
        REP SELECTOR ⟨field1⟩ ⟨attributes⟩.
          SELECTOR ⟨field2⟩ ⟨attributes⟩.
        ...
          SELECTOR ⟨fieldn⟩ ⟨attributes⟩:
        /* Other Declarations *
          ...
        * Initialization code for items of that type */
          ...
        OPERATION:
⟨function1⟩: FUNCTION (⟨parameterlist⟩):
        BEGIN:
        * Code for function 1 *
        END:
...
⟨function n⟩: FUNCTION (⟨parameterlist⟩):
        BEGIN:
        * Code for function n *
        END:
        END ⟨name⟩:
```

where ⟨...⟩ means a program variable name or attribute.

The statement OPERATION separates the two parts of an abstraction. The first half describes creating an excapsulated data object—its internal representation (SELECTOR fields) and its initialization, while the second half gives the implementation of the functions on these objects.

For example, a stack could be described as:

```
STACK: ABSTRACTION:
        REP SELECTOR TOP FIXED BINARY.
                /* Next stack location */
          SELECTOR STORAGE (100) FIXED BINARY:
                /* Size of stack = 100 */
        /* Initialization code */
        TOP = 0; /* stack is empty */
```

This states that each instance of type STACK has two components—a TOP field and a STORAGE field (an array of 100 elements). TOP is set to 0 whenever a stack is first allocated.

The functions which operate on these abstract types follow the OPERATION statement. For example, a PUSH function could be written as:

```
PUSH: FUNCTION (A, B);
      DECLARE A TYPE (STACK),
              B FIXED BINARY;
      BEGIN;
      A..TOP = A..TOP + 1;
      A..STORAGE (A..TOP) = B;
      END;
```

If a function returns a value, then a RETURNS clause can be used. For example, a PUSH function that retruns the new stack which is the result of pushing a value onto the old stack can be specified as:

```
PUSH: FUNCTION (A, B) RETURNS (TYPE (STACK));
```

Abstract objects look somewhat like PL/1 structures so syntactically they are referenced in a similar manner via the.. operator. Outside of the abstraction, the object is a "black box" with no allowable references to selectors TOP or STORAGE.

The TYPE attribute is used to declare objects of an abstract type. To use a stack in a program you would first declare it:

```
DECLARE MYSTACK TYPE (STACK);
```

In order to have the initialization code executed, the INITIAL attribute is used:

```
DECLARE MYSTACK TYPE (STACK) INITIAL (STACK);
```

and MYSTACK would be initialized by the STACK initialization routine (the setting of MYSTACK..TOP to 0). MYSTACK may only be passed as an argument to one of the functions in the stack abstraction, and its selectors may not be referenced outside of the stack abstraction. Since this parameter passing mechanism is overly restrictive, Section 3 will describe extensions to relax these restrictions somewhat.

While this syntax looks different from PL/1, for the average program, most of the statements will be in the standard language. Standard PL/1 procedures can be placed within abstractions. A procedure will be accessible within the abstraction while a function will be accessible outside the abstraction.

2.2 *Specifications*

Just having abstract types helps in program design; however, more can be done. An assertion was added to implement specifications. A user could add an input condition (much like Hoare-type axioms [12]) on the definition of a function.

For example, the PUSH function of the last section will be a legal operation if the stack is not full, i.e. TOP < 100, as in:

```
PUSH: FUNCTION (A, B);
      DECLARE A TYPE (STACK),
              B FIXED BINARY;
      PUSH: ASSERT (A..TOP < 100); [*]
      BEGIN;
      ...
      END;
```

When PUSH is called, the ASSERT is executed and the expression is evaluated. If false, the ASSERTFAIL condition is raised. The user can detect such a failure by includ-

* The string "PUSH:" is redundant here, and a later version of the system will not require it in this context.

ing an ASSERTFAIL ON block in the program, as in:

```
ON ASSERTFAIL BEGIN;
        * Do something *
        ...
        END;
```

The ONASSERT pseudovariable was added to return the label (or statement number if no label) of the assertion that failed. (The reason for the "PUSH:" in the above example.)

Asserts that remain true are specified as invariants. For example:

ASSERT (I < 100) INVARIANT;

will check that I is less than 100 at the start of every statement in the block containing the assert. This is similar to the UNDER construct in PLITS [13].

Since the language now handled assertions and their failures, specifications were added. Since the basic PL/1 ON block is too error prone and is easily used incorrectly, a separate EXCEPTION block in the declarative part of the abstraction was created. ASSERTs are used to determine whether the parameters to a function are within the domain for the function, and the EXCEPTION block is used to either handle the error condition if the arguments were outside of the domain, or to extend the domain to include the new arguments. To create exceptions, following the initialization description of an object, the user writes:

```
EXCEPTION;
        DO CASE (ONASSERT);
        \'PUSH'\ DO;
            * Process PUSH failure *
            ...
            END;
        \'POP'\ DO;
            * Process POP failure *
            ...
            END;
        ...
        END;
END;
```

(The DO CASE is a local addition to the PL/1 compiler to be described in Section 3. It is similar to the SELECT statement in other PL/1 implementations, and can be replaced by a series of IF ... ELSE IF .. ELSE IF ... ELSE ... constructs. Further details about the language are described in [14]).

If an ASSERT fails in an abstraction, the program automatically invokes the corresponding EXCEPTION block. The EXCEPTION block is a static property of the program and not dynamically set, like ON blocks. This is more amenable to verification and testing.

This implementation of abstractions and assertions shows the great similarity between the two major specification techniques: Hoare type axioms and Algebraic specifications [15]. Within an abstraction, the assertions behave as predicates obeying the rule P(X){F(X)}Q(X) for the program:

```
F: FUNCTION (X);
        ASSERT (P(X));
        BEGIN;
        Code for F(X);
        ASSERT (Q(X));
        END;
```

where P is the precondition to function F and Q is the postcondition.

Outside of the abstraction, the only assertions that can be written are relationships between the various functions since no accessing of the internal selectors is possible. Thus if PUSH and POP are written as functions that return a new stack after each call, as in:

```
PUSH: FUNCTION (A, B) RETURNS (TYPE (STACK));
  DECLARE A TYPE (STACK), B FIXED BINARY;
  ... /* details for PUSH */
POP:FUNCTION (A) RETURNS (TYPE (STACK));
  DECLARE A TYPE (STACK);
  ... /* details for POP */
```

then the assertion:

```
DECLARE X TYPE (STACK),
        Y FIXED BINARY;
ASSERT (POP (PUSH (X, Y)) = X);
```

corresponds to a runtime validation of the algebraic axiom that POP is an inverse operation to PUSH.

3. IMPLEMENTATION DESIGN

After designing the extensions of Section 2, we wanted to implement the additions within the PLACES Project at the University of Maryland [16]. PLACES contains the PLUM PL/1 compiler which compiles a large subset of the PL/1 language at a speed of several hundred statements per second on a Univac 1100/42 computer.

PLUM has a fairly standard structure: The first pass is driven by a parser which calls a scanner to return tokens, and builds an intermediate text. The second pass resolves symbol table addresses and discovers semantic errors. The third pass generates executable machine language for the program.

In order to implement the extensions, the initial idea was to extend the compiler to accept the new statements. However, this approach was not deemed cost effective. Since the extensions were of a research nature, they were very experimental and subject to frequent changes. The language extensions described in Section 2 is the result of several years of evolution as we clarified our ideas about abstraction. The compiler was too complex to be altered that often.

The alternative approach to build a preprocessor was also rejected. The cost of building a preprocessor to parse PL/1 seemed high, and the clumsy usage of it, as explained in Section 1, would mean that it would not get used very often. We could have forced students to use a preprocessor; however, we believed that such usage would be artificial and not representative of typical usage of a compiler.

The approach that was used was to divide the extensions into two parts—those primitive features which had to be added to the PLUM compiler and those features which could easily be simulated by the existing PL/1 language. It turns out that very few of the features of Section 2 are actually primitive, and the extensions to the compiler to implement these went very quickly.

3.1 *Compiler extensions*

As will be shown in Section 4, the implementation of abstract data types depends on POINTER variables, which were already implemented in PLUM. The major extensions needed to the compiler were in the handling of the TYPE attribute. For a variable X declared TYPE(THING), the symbol table manager created an entry for X of type POINTER, and the code generator was altered so that X could be only passed as an argument to an entry point within the procedure THING. The actual syntax was TYPE(A:B, C, D, ...) and X could only be passed to one of the procedures named in the TYPE capability list (A, B, C, D, ...) [16]. This allowed types to be passed to several different abstractions.

In addition, accessing components within X (e.g., X..FIELD1) could only occur within the procedure THING, or in the general case of TYPE(A:B, C, D) only within A. Thus even though variables of TYPE(A) could be passed to procedures B, C, and D, only A knew the internal representation of the object and could manipulate the internal fields. This added the necessary protection.

In order to do assignment to abstract types, the assignment operator had to be modified since A = B would simply set pointer A to be the same as pointer B, whereas the value pointed to by B is to be copied into the object pointed to by A. This was handled by the addition of the := assignment. If the operands to this assignment are native PL/1 variables, then := means the same as =. If the operands are TYPE variables, then the correct copy operation is applied.

In the PLUM implementation, when exiting a procedure containing a TYPE (i.e. POINTER) variable, the storage pointed to by the variable is returned to the system if nothing else points to this storage. For a TYPE variable this will be true—they are effectively PL/1 automatic storage. Thus garbage collection of dynamic storage is handled automatically. In a production PL/1 system this could be handled by including a FREE statement for each TYPE variable at procedure exit.

For assertions, the statement

ASSERT (expression)

was compiled as if the following were coded:

IF (expression) THEN SIGNAL ASSERTFAIL;

ASSERTFAIL had to be added to the list of PL/1 ON conditions, and ONASSERT had to be added as a pseudovariable. All of these were relatively straightforward additions to the compiler.

The statement

ASSERT (expression) INVARIANT;

did require some additional code generation considerations since the expression had to be tested between every statement in the current block. This was done by implementing the ASSERT as an internal subroutine that was called between every statement of the block.

That is, INVARIANT generates the code:

```
   GOTO L;
ASSERT: "PROCEDURE";
     ASSERT (expression);
   END ASSERT;
L:...
```

where "PROCEDURE" stands for an internal procedure (e.g., no activation record or dynamic storage is needed and control passes back to the caller with a minimum of overhead) and the code

"CALL" ASSERT;

was added to the code generated at the start of each statement in the current block. Similar to "PROCEDURE", "CALL" is an efficient linkage to an internal subroutine. Although not the most efficient implementation, the code is correct and adequate in a developmental environment.

3.2 *Macro extensions*

With the above additions, a macro processor within PLUM was designed. Parsing in PLUM is relatively straightforward, with the parser calling the scanner for successive tokens (Fig. 1). Each of the special terms (e.g. ABSTRACTION, REP, etc.) was made a

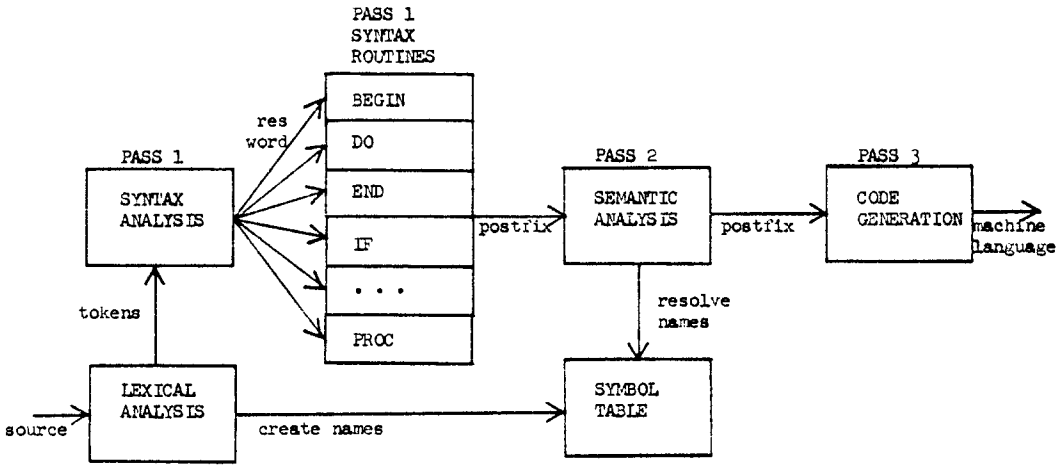

Fig. 1. Structure of PLUM PL 1 compiler.

reserved word by simply adding it to the symbol table of the compiler. The parser calls a unique function for each reserved word. The functions for each of these new reserved words simply redirect the scanner to return a table of successive tokens instead of reading the regular source input file.

For example, ABSTRACTION was implemented by creating a procedure which returns a pointer value. Thus ABSTRACTION resets the scanner to point to the table:

```
PROCEDURE
RECURSIVE
RETURNS
(
POINTER
)
⟨end of table⟩
```

ABSTRACTION substitutes this list for itself on the following seven calls to the scanner. Thus the parser is unaware that anything but a legal procedure is being declared. (The code substituted for ABSTRACTION is slightly modified from the above. The actual code is given in Section 4.)

In order to fully implement this macro processor, the following set of commands are implemented:

MCSTRT. Start scanner to point to next command as next operation (token to return).

MCEND. Turn off macro table. Scanner should start to read source input again as the next token to return.

MCTOKN (X, Y). Return a token of class Y and value X. Thus MCTOKN (ACRCUR, LXKYWD) returns the token for RECURSIVE (a keyword (LXKYWD) with the internal value ACRCUR).

MCGET. This creates a new symbol table name and returns it. It is used to create declarations of internal variables not known to the programmer.

MCSETT X. This sets condition X to be true. X is a boolean variable. For example, MCSETT EXCEPTION is used in the code for EXCEPTION to insure that only one EXCEPTION block appears in a program. If the named condition has already appeared, then an error results.

MCSETF X. The named condition X is initialized to false. This is used in conjunction with the MCSETT command.

MCREF. Return as next token a previously generated name. Thus the implicitly declared variable (via MCGET) can be referenced in later statements.

MCSUB X. Call subroutine X to perform some additional processing. This is an "escape" from the macro processor back into the environment of the parser. As it turned out, this was only used to generate error messages, which could have easily been implemented as a separate MCERROR macro. (See next section.)

MCIF X. Go to next table entry if boolean X is true. If not, skip to the MCENDIF (or MCELSE) entry. This enabled conditional code to be generated depending upon the source program.

MCENDIF. Terminates the preceding MCIF block.

MCELSE. Allows for separate true and false code to be generated based upon the condition in the MCIF command. For this particular implementation, this was never needed.

With these commands, it was now possible to implement the extensions as expansions of existing PL/1 statements. In Section 4, the resulting tables will be given. As an interesting sidelight to the implementation, one of the hardest problems we faced was keeping source statement numbers correct. An important engineering aspect of the implementation was that users should have no knowledge or idea of the underlying implementation. These language extensions should appear to be an integral part of the language, as much as the DECLARE statement or assignment statement. Thus we wanted to make sure that statement numbers corresponded to the source statement numbers in the input file for the program. Not having PLUM increment the current statement counter when a macro generated statement was parsed turned out to be relatively difficult.

3.3 *Error detection*

An important issue in compiler design, especially for PLUM where good diagnostics are a requirement in a university environment, is error detection. In adding abstractions, errors were handled in one of three ways:

(a) Do nothing
(b) Modify compiler to check for error
(c) Let macros check for errors

Macros simply substituted parsed text for a given symbol. Thus if an error occurred, the "correct" syntax fed to PLUM would be in error and a message would be generated by the parser. Thus as much as possible, option (a) was the desired choice. For example, if a program contains the statements:

```
PUSH: FUNCTION;
```

and

```
CALL PUSH (X, Y);
```

then the compiler would generate the message that the arguments to PUSH do not agree with its definition, e.g. the message is independent of abstraction implementation.

Unfortunately, not every error can be handled intelligently in this manner. (The crucial word is "intelligently", since the method will find all errors.) For example, multiple EXCEPTION blocks would generate the message that procedure EXCEPTION is multiply defined, yet the user has no knowledge about any procedure named "exception".

The macros themselves can handle most of these errors. The MCSETT (set to true) and MCSETF (set to false) commands of the last section set global flags. These can be tested by the MCIF command. Thus OPERATION sets a flag that is tested by FUNCTION to make sure OPERATION appeared. EXCEPTION sets a flag that is tested by FUNCTION to determine whether to generate code to raise an exception (ON unit) if an assert failure occurs.

In order to handle the remaining errors, a few tests had to be added to the parser itself. For example, ABSTRACTION can only appear as a level one (outermost) procedure. Thus the parser first checks the nesting level before processing the abstraction macro.

4. IMPLEMENTATION

As mentioned previously, POINTER variables form the basis for the abstract data type. The user wants something of TYPE(THING), so the system implements this as a POINTER, with the constraint that the object is only passed as an argument to the procedure THING.

Within the abstraction, the abstract object is a pointer variable. The representation (REP) is a BASED structure referenced by the POINTER variable. Code is automatically generated to allocate storage for such a based structure and to initialize the abstract variable to point to this structure. The abstraction selector operator (..) was chosen because of its similarity to the syntactic structure accessing operator (.); however, it is actually a pointer reference (->). While its semantics are similar to structure accessing, its implementation is very different.

In order to allow users to also allocate abstract types, e.g. to make linked lists, the function NEW was added. X = NEW; allocates a new copy of the object TYPE(THING) and sets X to point to it. This is similar to NEW in Pascal. NEW can be used with any abstract data type, and is implemented as a local procedure within the abstraction procedure defining the abstract type.

4.1 *Macro tables*

ABSTRACTION. The basic abstraction is a procedure that returns a pointer to the allocation of the abstract object. This is handled by the code:

```
$1: PROCEDURE RECURSIVE RETURNS (POINTER);
  DECLARE $2 POINTER;
  $2 = NEW;
```

where $1 is the name of the abstraction and $2 is a generated dummy name.

If a user declares a variable to be of TYPE ($1), then abstraction procedure $1 is called in the INITIAL clause of the declaration and the value of pointer $2 will be returned via the later OPERATION clause. Note that following ABSTRACTION, the executing program has already allocated $2. Any initialization code will set the appropriate values pointed to by $2 before returning the initial allocation. As an example of a complete macro, the details for ABSTRACTION are given in Fig. 2.

OPERATION. OPERATION has the primary function of returning the value allocated via $2 = NEW for the above ABSTRACTION. Syntactically it follows any initialization code so that the allocated object can be initialized "automatically". OPERATION results in the followikng code being generated:

```
RETURN ($2); /* RETURN VARIABLE OF TYPE ($1) */
```

```
MCSTRT                      Start macro processor
MCTOKN  ACRCUR,LXKYWD       RECURSIVE
MCTOKN  ACRTNS,LXKYWD       RETURNS
MCTOKN  ACLPAR,LXLP         (
MCTOKN  ACTYPE,LXATTR       TYPE
MCTOKN  ACLPAR,LXLP         (
MCREF   1                   abstraction name $1
MCTOKN  ACRPAR,LXRP         )
MCTOKN  ACRPAR,LXRP         )
MCTOKN  ACSEMI,LXRSWD       ;
MCTOKN  ACDCL,LXRSWD        DECLARE
MCGEN                       dummy name $2
MCTOKN  ACTYPE,LXATTR       TYPE
MCTOKN  ACLPAR,LXLP         (
MCREF   1                   abstraction name $1
MCTOKN  ACRPAR,LXRP         )
MCTOKN  ACSEMI,LXRSWD       ;
MCREF   $2                  dummy name $2
MCTOKN  ACEQ,LXBN           =
MCREF   4                   NEW (allocation function)
MCSETF  EXCEPTION           No exception block yet
MCSETF  OPERSEEN            No OPERATION seen
MCSETF  REPSEEN             No REP seen yet
MCSETT  NOOPER              No OPERATION yet
MCSETT  NOREP               No REP yet
MCEND                       That's all for ABSTRACTION
```

Fig. 2. Macro table for ABSTRACTION.

```
NEW: PROCEDURE RETURNS (POINTER);
      * ALLOCATION PROCEDURE *
      DECLARE $3 POINTER;
      ALLOCATE $1 SET ($3);
      RETURN ($3);
      END; * NEW *;
```

Note that the allocate statement allocates a BASED structure ($1) and sets a pointer ($3) to it.

REP and SELECTOR. REP defines the actual data structures used by the abstraction. The name of the based structure will be the same as the abstraction name. The code substituted for REP is:

```
DECLARE 1 $1 BASED($2).
```

The expression BASED($2) is included so that initialization will work as expected, i.e. $2 will be the assumed pointer if any selector fields are referenced. The statement TOP = 0 means the same as the statement $2 -> TOP = 0.

SELECTOR simply returns the value 2. Thus a REP statement is compiled as:

```
DECLARE 1 abstracttype BASED($2).
          2 field1 attributes,
          2 field2 attributes,
          . . . ;
```

EXCEPTION. EXCEPTION creates a procedure that gets called by ON statements at each FUNCTION entry. The code added is:

```
EXCEPTION: PROCEDURE;
```

Set EXCEPTION flag to true;

The END following the EXCEPTION block terminates this procedure.

FUNCTION. Functions are simply entry points into the abstraction procedure. The code subsituted is:

```
entryname: ENTRY(parameters);
```

if EXCEPTION flag is true then generate:

```
ON ASSERTFAIL CALL EXCEPTION;
```

FUNCTION was the one statement that did require some changes to the compiler. FUNCTION was implemented by substituting the token ENTRY and setting a flag, checked by the parser, to call the macro processor after the parameter list was processed.

This change would be sufficient for correct program development, but does not include all the necessary enforcement mechanisms to detect errors. Thus several additional changes had to be made:

(1) Only BEGIN or DECLARE statements can follow FUNCTION statements. The body of each function is then a separate block of code with its own activation record.
(2) GOTOs between functions are not permitted. At run time, any GOTO which results in a function's BEGIN block being popped off the execution stack is processed as an error.
(3) The END statements for functions are handled in the same manner as procedure END statements, i.e. a RETURN is added. This (and the above GOTO restriction) forces each function to be an independent subroutine with a unique entry point, and a unique set of return points. Common operations among functions in an abstraction can be handled by internal procedures inside the abstraction.

```
        RETURN(A..TOP=0);
    END;
    END STACK;
STACK: ABSTRACTION;
REP SELECTOR STORAGE(STACKSIZE) FIXED BINARY,
     SELECTOR TOP           FIXED BINARY,
     SELECTOR SIZE          FIXED BINARY;
 /* Set Initial stack size */
     DECLARE STACKSIZE STATIC INIT(100);
/* Initialization code */
     TOP = 0;
     SIZE = STACKSIZE;
     EXCEPTION;
           DO CASE(ONASSERT);
           \'PUSH'\ DO;
                DECLARE NEWSTACK TYPE(STACK),
                        I FIXED BINARY;
              /* Increase stack size due to overflow */
              STACKSIZE = A..SIZE +100;
              NEWSTACK = NEW;
              /* Copy old to new */
              DO I = 1 TO A..SIZE;
                      NEWSTACK..STORAGE(I)=A..STORAGE(I);
                      END;
              NEWSTACK..TOP = A..TOP;
                /* Set user stack to be new one */
              A := NEWSTACK;
              /* Set STACKSIZE for next default allocation */
              STACKSIZE = 100;
                END;
           \'POP'\ DO;
                /* Print message and return 0 */
                PUT SKIP LIST('*** STACK EMPTY. 0 RETURNED');
                A..TOP = 1;
                A..STORAGE(A..TOP) = 0;
                END;
           /* No exception for EMPTY */
           END; /* DO CASE */
    END; /* EXCEPTION */
     OPERATION;
PUSH: FUNCTION(A,B);
     DECLARE A TYPE(STACK),
             B FIXED BINARY;
     PUSH: ASSERT (A..TOP < A..SIZE);
     BEGIN;
          A..TOP = A..TOP +1;
          A..STORAGE( A..TOP ) = B;
     END;
POP: FUNCTION(A,B);
     POP: ASSERT (A..TOP > 0);
     BEGIN;
          B = A..STORAGE( A..TOP );
          A..TOP =  A..TOP -1;
     END;
EMPTY: FUNCTION(A) RETURNS(BIT);
     BEGIN;
```

Fig. 3. Abstraction module.

In Fig. 3, a complete stack implementation is given. In this implementation, stacks are allowed to grow arbitrarily large, yet the implementation preserves the efficiency of the array implementation (except in the rare cases of the stack overflowing). The EXCEPTION block is used to enforce the specifications on the stack functions, and to extend the domain of the PUSH function by enlarging the stack when a specification failure occurs (stack overflow). Figure 4 gives the code as compiled by PLUM to implement this stack. Macro reserved words are given as comments and the PL/1 syntax substituted for them is underlined.

5. RESULTS OF IMPLEMENTATION

The system has been used in several and fairly heavily in one class at the University of Maryland. Students were not fully aware of the underlying implementation of abstractions—probably the best measure of its success as an implementation technique. Only 29% assumed that the partial macro substitution technique was used.

The language that we developed seemed successful, given our limited usage. In one experiment, two classes had to write scanners for a compiler. One class used abstractions while the other used standard PL/1. In each case, the final product was run against a standard test data set, and only those programs that worked correctly were evaluated. This resulted in 8 programs from each class. The programs were then run with all of the PLACES diagnostic features turned on (e.g. static and dynamic statement counts, timing information, tracing, etc.) Some of the results are presented in Fig. 5.

```
STACK: /* ABSTRACTION */ procedure recursive returns(pointer);
     declare dummy pointer;
     dummy = new;
/* REP */ declare 1 stack based(dummy),
     /* SELECTOR */ 2 STORAGE(STACKSIZE) FIXED BINARY,
     /* SELECTOR */ 2 TOP           FIXED BINARY,
     /* SELECTOR */ 2 SIZE          FIXED BINARY;
/* Default stack size */
     DECLARE STACKSIZE STATIC INIT(100);
/* Initialization code */
     dummy -> TOP = 0;
     dummy -> SIZE = STACKSIZE;
     EXCEPTION: procedure(onassert);
          DO CASE(ONASSERT);
          \'PUSH'\ DO;
               DECLARE NEWSTACK TYPE(STACK),
                       I FIXED BINARY;
             /* Increase stack size due to overflow */
               STACKSIZE = A /* .. */ -> SIZE +100;
             NEWSTACK = NEW;
             /* Copy old to new */
             DO I = 1 TO A /* .. */ -> SIZE;
                     NEWSTACK /* .. */ -> STORAGE(I)=A /* .. */
                              -> STORAGE(I);
                     END;
             NEWSTACK /* .. */ -> TOP = A /* .. */ -> TOP;
               /* Set user stack to be new one */
             A -> STACK = NEWSTACK -> STACK;
             /* Set STACKSIZE for next default allocation */
             STACKSIZE = 100;
               END;
          \'POP'\ DO;
               /* Print message and return 0 */
               PUT SKIP LIST('*** STACK EMPTY. 0 RETURNED');
               A /* .. */ -> TOP = 1;
               A /* .. */ -> STORAGE(A /* .. */ -> TOP) = 0;
               END;
          /* No exception for empty */
          END; /* DO CASE */
    END; /* EXCEPTION */
/* OPERATION; */
     return(dummy);
     /* allocation function */
     new: procedure returns(pointer);
          declare initptr pointer;
          allocate stack set(initptr);
          return(initptr);
          end new;
PUSH: /* FUNCTION */ entry(A,B);
     on assertfail call exception(onassert);
     DECLARE A /* TYPE(STACK) */ pointer,
              B FIXED BINARY;

     BEGIN;
          A /* .. */ -> TOP = A /* .. */ -> TOP +1;
          A /* .. */ -> STORAGE( A /* .. */ -> TOP ) = B;
          return;
     END;
POP: /* FUNCTION */ entry(A,B);
     on assertfail call exception(onassert);
     POP: ASSERT (A /* .. */ -> TOP > 0);
     BEGIN;
          B = A /* .. */ -> STORAGE( A /* .. */ -> TOP );
          A /* .. */ -> TOP = A /* .. */ -> TOP -1;
     return;
     END;
EMPTY: /* FUNCTION */ entry(A) RETURNS(BIT);
     on assertfail call exception(onassert);
     BEGIN;
          RETURN(A /* .. */ -> TOP=0);
     END;
     END STACK;
```

Fig. 4. Code compiled for abstraction.

Feature	Abstractions Mean	Abstractions Std-Dev	Standard PLUM Mean	Standard PLUM Std-Dev
Statements	281	60	321	30
Statements executed	17597	6928	23868	11645
Object pgm size(words)	2554	784	2888	347
Object pgm words/Stmt	9.1	2.3	9.0	1.8
IF statements	19.25	6.56	34.75	15.06
IF stmts executed	3059	2182	7070	6301
% IF stmts executed	16.38	5.23	27.61	18.47
PROC statements	15.76	1.85	12.00	3.70
PROC stmts executed	1976	1299	1629	798
% PROC stmts executed	10.66	3.18	7.01	2.29

Fig. 5. Collected data.

Both groups had comparable products, but the abstraction group used only an average of 281 statements as compared to the 321 of the standard PL/1 group. This seemingly contradicts the objection against abstractions that they lead to larger programs; however, our results were not statistically significant.

According to Fig. 5, the abstraction group executed fewer statements, contained significantly fewer IF statements, but executed more procedure statements. This agrees with our preconceived notions that abstractions are less complex (as measured by number of IF statements), but incur greater overhead due to information hiding aspects of abstract data type design.

While the overhead for procedure calls was significant, many of the procedures could be classified as:

(a) Trivial—simply sets or returns a value of one of the selectors in the implementation of the abstract type
(b) Simple—a one line function (e.g. EMPTY in Figure 3) In our sample data, rewriting these as inline functions reduced the procedure calls to 1696 (compared to 1629 of the standard group), which is much less additional overhead. A "smart" compiler (or an INLINE attribute) would allow such optimization without violating the "black box" approach towards abstractions.

SUMMARY

In this paper we have outlined the procedure for extending a compiler with new features by using a form of macro-processor hidden from the user. The user is unaware of any changes to the source program and simply views the new concepts as extensions to the basic language.

The implementation of data abstractions via macro substitutions within PLUM has been straightforward and efficient. Only minor changes had to be made to either the parser or the code generator of PLUM, and once implemented, the macro-processor could be easily altered. A few primitive constructs were added to the compiler, and the remainder of the additions were handled by a macro processor built into the scanner. Therefore, users were only aware of the original source program and the system was efficient since only one parsing pass was needed to process the program.

Data was collected from two classes—one using abstractions and one using standard PL/1. Programs written with abstractions turned out to be less complex and with more procedure invocations (as theorized) but contained fewer statements on the average (a surprizing, but favorable result).

The implementation technique that we have described seems quite practical in a research or development environment. Although the process generates more object code than a production compiler especially tuned for the new statements, in most applications such tight coding is not needed. The versatility of being able to extend the compiler fairly easily is probably worth the cost.

REFERENCES

1. ACM Language design for Reliable Software Conference, Raleigh, NC, *Sigplan Notices* **12**, (1977).
2. D. Schwabe and C. J. Lucena, Design and implementation of a date abstraction definition facility, *Software Practice and Experience* **8**, 709–719 (1978).
3. B. Leavenworth, Extended PL/1 Reference Manual, Technical Memo 19, IBM T. J. Watson Research Center (1979).
4. M. Stonebraker *et al.*, The design and implementation of INGRES, *ACM Trans. Database Syst.* **1**, 189–222 (1976).
5. W. Wulf, R. London and M. Shaw, An introduction to the construction and verification of Alphard programs, *IEEE Trans. Software Engng* **2**, 253–264 (1976).
6. G. J. Popek, Notes on the design of Euclid, ACM Language Design for Reliable Software, *Sigplan Notices* **12**, (1977).
7. A. L. Ambler *et al.*, Gypsy: A language for specification and implementation of verifiable programs, ACM Language Design for Reliable Software, *Sigplan Notices* **12**, (1977).
8. Department of Defense, Reference Manual for the Ada Programming Language-Proposed Standard Document (1980).
9. B. Liskov *et al.*, Abstraction mechanisms for Clu, *Commun. ACM* **20**, 564–576 (1977).

10. D. L. Parnas. On the criteria for decomposing systems into modules. *Commun. ACM* **15**, 1053–1058 (1972).
11. M. V. Zelkowitz. PLACES: Programming language and construct evaluation system. NBS and ACM Seventeenth Annual Technical Symposium, Gaithersburg MD, pp. 79–85 (1978).
12. C. A. R. Hoare, An axiomatic basis for computer programming. *Commun. ACM* **12**, 576–580, 583 (1969).
13. J. A. Feldman. High level programming for distributed computing. *Commun. ACM* **22**, 353–367 (1979).
14. M. V. Zelkowitz and J. Lyle. Implementation of program specifications. IEEE Computer Society COMPSAC 80. Chicago, Illinios, pp. 194–200 (1980).
15. J. Guttag. Abstract data types and the development of data structures. *Commun. ACM* **20**, (1977).
16. M. V. Zelkowitz and H. J. Larsen. Implementation of a capability based data abstraction. *IEEE Trans. Software Engng* **4**, 56–64 (1978).

About the Author—MARVIN V. ZELKOWITZ. Associate Professor of Computer Science at the University of Maryland, was born in Brooklyn, New York. He obtained a B.S. in Mathematics (1967) from Rensselaer Polytechnic Institute and an M.S. (1969) and a Ph.D. (1971) in Computer Science from Cornell University. He has been at the University of Maryland since 1971. His current interests are in programming language design and implementation and in programming environments. He is a senior member of the IEEE Computer Society and a member of the Association for Computing Machinery. He is the past chairman of ACM's Special Interest Group of Software Engineering (SIGSOFT) and a recent chairman of the Computer Society's Washington DC chapter.

About the Author—JAMES R. LYLE was born in Washington, DC. He obtained a B.S. in Mathematics from East Tennessee State, an M.S. in Mathematics, and is completing the requirements for a Ph.D. degree in Computer Science from the University of Maryland. He is interested in programming language design and compiler implementation. He is a member of the Association for Computing Machinery and of the IEEE Computer Society.