MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 88

An Approach to Abstraction

by

Barbara Liskov

Stephen Zilles*

Work reported herein was supported in part by the National Science Foundation under research grant GJ-34671.

*Cambridge Systems Group, IBM Systems Development Division.

September 1973

FOREWORD

This memo describes a preliminary version of a structured programming language being developed within the base language group. Although the basic form of the language is fixed (its Pascal-like syntax and its dependence on function clusters), the details of its syntax and semantics can be expected to change in the near future. In addition, the language as described in this memo is by no means complete: missing features include parallelism and error handling.

Abstract

The motivation behind the work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program will be more reliable as a result. Clearly, this is a worthwhile goal.

Unfortunately, it is very difficult for a designer to select in advance all the abstractions which the users of his language might need. If a language is to be used at all, it is likely to be used to solve problems which its designer did not envision, and for which the abstractions embedded in the language are not sufficient.

This paper presents an approach which allows the set of built-in abstractions to be augmented when the need for a new abstraction is discovered. This approach to the handling of abstraction is an outgrowth of work in designing a language for structured programming. Relevant aspects of this language are described, and examples of the use and definitions of abstractions are given.

Introduction

This paper describes an approach to computer representation of abstraction. The approach, developed while designing a language to support structured programming, is also relevant to work in very-high-level languages. We begin by explaining its relevance and by comparing work in structured programming and very-high-level languages.

The purpose of structured programming is to enhance the reliability and understandability of programs. Very-high-level languages, while primarily concerned with increasing programmer productivity by easing the programmer's task, can also be expected to enhance the reliability and understandability of code. Thus, similar benefits can be expected from work in the two areas.

Work in the two areas, however, follows very different lines of approach. A very-high-level language attempts to present the user with the abstractions (operations, data structures, and control structures) useful to his application area. The user can use these abstractions without being concerned with how they are implemented -- he is only concerned with what they do. He is thus able to ignore details not relevant to his application area, and to concentrate on solving his problem. Structured programming, on the other hand, attempts to impose a discipline on the programming task so that the resulting programs are "well-structured."

In structured programming, a problem is solved by means of a process of successive decomposition. The first step is to write a program which solves the problem but which runs on an abstract machine, i.e., one which provides just those data objects and operations which are suitable to solving the problem. Some or all of those data objects and operations are truly abstract, i.e., not present as primitives in the programming language being used. We will, for the present, group them loosely together under the term "abstraction."

The programmer is initially concerned with satisfying himself (or proving) that his program correctly solves the problem. In this analysis he is concerned with the way his program makes use of the abstractions, but not with any details of how those abstractions may be given concrete definitions. When he is satisfied with his program, he then turns his attention to the abstractions it uses. Each abstraction represents a new problem, requiring additional programs for its solution. Again the new program may be written to run on an abstract machine, introducing further abstractions. The original problem is completely solved when all abstractions generated in the course of providing the program have been realized by further programs.

It is obvious now that the approaches of very-high-level languages and structured programming are not so dissimilar as it first appeared, since at the core of each approach is the idea of making use of the abstractions which are correct for the problem being solved. Furthermore, the rationale for using the abstractions is the same in

the two approaches: to free the programmer from concern with details not relevant to the problem he is solving.

In most very-high-level languages, the designers attempt to identify the set of useful abstractions in advance. A structured programming language, on the other hand, contains no preconceived notions about the particular set of useful abstractions but, instead, must provide a mechanism whereby the language can be extended to contain the abstractions which the user requires. A language containing such a mechanism can be viewed as a general-purpose, indefinitely-high-level language.

This paper describes an approach to abstraction which permits the set of built-in abstractions to be augmented when the need for new abstractions is discovered. The paper also describes the realization of the approach in a programming language (developed to support structured programming) and gives some examples of its use. Remaining sections of the paper discuss the relationship of the approach to previous work, and some aspects of the implementation of the language.

The Meaning of Abstraction

The definition of structured programming given in the preceding section is quite vague because it is couched in such undefined terms as "abstraction" and "abstract machine". In this section we will provide the definition for "abstraction" which underlies the structured programming language we are developing.

What we desire from an abstraction is a mechanism which permits the expression of relevant details and the suppression of irrelevant details. If we consider conventional programming languages, we discover that they offer one powerful aid to abstraction: the function or procedure. When a programmer makes use of a procedure, he is (or should be) concerned only with what it does -- what function it provides for him. He is not concerned with the algorithm executed by the procedure. In addition, procedures provide a means of decomposing a problem -- performing part of the programming task inside a procedure, and another part in the caller of the procedure. Thus, the existence of procedures, and the possibility of separate compilation of procedures, goes quite far toward capturing the meaning of abstraction.

Unfortunately, procedures alone do not provide a sufficiently rich vocabulary of abstractions. In the definition of structured programming given in the previous section, we spoke about an abstract machine providing abstract data objects and operations. Procedures correspond only to abstract operations. To fully support abstractions, a programming language must provide abstract data objects as well. It is neither realistic nor helpful, however, to consider abstract objects and operations separately. Obviously, objects and operations are intimately related. Each operation may be applied only to certain objects, and each object may only be operated on by certain operations. A structured programming language must allow the relationships between objects and operations to be expressed.

The leads us to the characterization of an abstract data type which is central to the design of the language. The most important form of abstraction is provided by an abstract data type whose characteristics are completely defined by the set of operations available on that type.

Abstract types are intended to be very much like the primitive types provided by a programming language. The user of a primitive type is only concerned with creating (declaring) objects (variables) of that type and then performing operations on them. He is not (usually) concerned with how the data objects are represented, and he views the operations on the objects as indivisible and atomic when in fact several machine instructions may be required to perform an operation. In addition, he is not (in general) permitted to decompose primitive-type objects. Consider, for example, the primitive type integer. A programmer wants to declare objects of type integer and to perform the usual arithmetic operations on them. He is usually not interested in an integer object as a bit string, and cannot make use of the format of the bits within a computer word. Also, he expects the language to protect him from foolish misuses of types (e.g., adding an integer to a character) either by treating such a thing as an error (strong typing), or by some sort of automatic type conversion.

In the case of a primitive data type, like integer, the programmer is making use of a concept or abstraction which is realized at a lower level of detail -- the programming language itself and its

compiler. Similarly, an abstract data type is used at one level and realized at a lower level, but the lower level does not come into existence automatically by being part of the language. Instead, an abstract data type is realized by writing a certain kind of program which defines the type in terms of the operations which can be performed on it. The language facilitates this activity by allowing the use of an abstract data type without requiring its on-the-spot definition. The language processor supports abstract data types by building links between the use of a type and its definition (which may be provided either earlier or later) and by enforcing the view of a data type as equivalent to a set of operations by a very strong form of data typing.

Using Abstract Data Types

We will now present an example in order to give substance to the above discussion of abstract data types and related operations. True to the principle of structured programming, we will start at the top and will therefore discuss how to make use of abstractions before we discuss how to define them.

We have chosen for our example the following problem: Write a translator from an infix language to a Polish post-fix language. The translator is to be a general-purpose program which makes no assumptions about input or output devices (or files). It makes only the following assumptions about the input grammar:

1) The input language has an operator precedence grammar.

2) The symbols of the input language are identifiers, which are arbitrary strings of letters and numbers, and operators; blanks terminate symbols but are otherwise ignored.

We may restate the above problem description. The translator is a function of three arguments: an input file containing the sentence of the input language, an output file to accept a sentence of the output language, and a grammar to recognize symbols of the input language and determine their precedence relations. When the translator returns, the sentence in post-fix notation is in the output file.

We have chosen this problem as our example for the following reasons:

1. Both the problem and its solution are very familiar to people interested in programming languages; therefore, limited explanations are required.

2. The problem is nevertheless complex, and its solution makes use of many abstractions.

The program "Polish_gen" is shown in Figure 1. The language in which it is expressed is conventional in most respects. It draws heavily on PASCAL(1), but is modified as necessary to accommodate abstract data types. The modifications occur exactly where you would expect them: in data declarations, in performing operations on objects of abstract data type, and in defining abstract data types.

```
Polish_gen: procedure(g:grammar, input:infile, output:outfile);

 s: stack(token, grammar$eof_token(g));
 t: token(g);
 mustscan: Boolean(true);

 while ¬stack$empty(s) do
    if mustscan
       then t:=scan(input,g);
       else mustscan:=true;
    if token$is_op(t)
       then
          case grammar$prec_rel(stack$top(s),t,g) of
             "<":: stack$push(s,t);
             "=":: stack$erasetop(s);
             ">":: begin
                      outfile$out_str(output,
                         token$symbol(stack$pop(s)));
                      mustscan:=false;
                   end
          otherwise error;
       else outfile$out_str(output,token$symbol(t));
 end
 outfile$close(output);
 return;

 end Polish_gen
```

Figure 1

Interesting features of the language include:

1. Like PASCAL, the language is procedure oriented, and begin, end are used only to group statements.

2. Both primitive data types and data structuring methods are taken from PASCAL. An example of the former is the use of a data type type. The most important data structuring method is the record which defines an aggregate of sub-elements or components. Component definitions

   <selector-name> : <component-type>

   are separated by semicolons, and the whole record definition is

enclosed in parentheses. The components of a record are referenced using an infix dot notation which is left associative; for example,

s.x

means select component with <selector-name> x from record s.

3. The language uses a standard format for declarations:

<variable-name> : <description>

is used no matter what kind of variable is being declared.

4. The language borrows two things from PL/I: Use of

returns <type-list>

to specify the type of the value(s) returned from a procedure, and elimination of begin wherever possible, by assuming more than one statement will ordinarily follow a reserved word such as do and procedure.

5. The language has no goto statement and no labels.

6. A structured error-handling mechanism is under development. At present, it is indicated only by the presence of the reserved word error.

7. The language has no free variables in the standard sense. It does permit the free use of procedure names and data-type names.

The same syntax is used to declare objects of abstract data type as is used to declare objects of primitive (non-structured) type:

<var-name>: <type-name> [(<type-parameters>)]

<type-name> may name either a primitive type of the language or an abstract type. The <type-parameters>, which contain information used in creating instances of data, are optional as is indicated by the use of braces "[ ]" around the syntactic category.

An example of the use of a primitive type is in the line

mustscan: boolean(true)

which creates an object of type boolean with initial value true and assigns it to variable mustscan. The line

s: stack(token, grammar$eof_token(g))

declares s to be the name of a variable which holds an object of abstract type stack. It also illustrates the two uses for the <type-parameters>. The first parameter, the abstract type token, defines the type of element which may be placed on the stack s. The second parameter is an expression which when evaluated returns a value of type token which is then used to initialize the stack.

An explicit initial value is provided for almost every variable declared in the example. There are two reasons for this. First, we believe every variable should be initialized (at least to an error-causing undefined value) to prevent indeterminancy from accidentally occurring when a variable is used before it has been the target of an assignment. Second, it is frequently the case (e.g., with the stack s) that a variable must be initialized to begin an iterative process and that initialization is more efficient when combined with allocating the variable than it is when a separate assignment is used.

The compiler for the language is prepared to encounter <type-name>s which are not in its repertoire of primitive types. It assumes the meaning of such names will be provided by a piece of the program which will be (or has been) separately compiled. The way in which the compiler attaches a meaning to an abstract <type-name> will be discussed in a later section.

All of the formal parameters and some of the local variables of Polish_gen are defined as abstract types. This raises the question of how can an abstract object be used in Polish_gen or, in fact, any procedure? There are three ways in which an abstract object can be used:

1. Abstract objects may be passed as indivisible entities between procedures. In this case, the type of the actual argument passed by the calling procedure must be identical to the type of the corresponding formal parameter in the called procedure.

2. Abstract objects may be stored away, but only in variables declared to hold objects of that type.

3. Abstract objects may be operated upon by the operators which characterize the abstract type.

The statement

```
t:=scan(input, g)
```

illustrates the first two kinds of usage. The procedure scan, shown

```
scan: procedure(input:infile, g:grammar) returns token;

   newsymb: string(null);
   ch: char(" ");

   while ch=" " do ch:=infile$get(input); end
   if infile$eof(input) then return(grammar$eof(g));
   if letter(ch) or number(ch) then
      begin
         while letter(ch) or number(ch) do
            newsymb:=newsymb concat ch;
            ch:=infile$get(input);
         end
         infile$putback(input,ch);
      end
      else newsymb:=unit_string(ch);
   return token(g,newsymb);

   end scan
```

Figure 2

in Figure 2, expects objects of type infile and grammar as its arguments, and returns an object of type token, which is then stored in the token variable t. Because scan is not one of the characterizing operations for either type infile or type grammar, it must treat its arguments as indivisible objects. It does, however, perform operations on those objects.

Applications of a characterizing operation to an abstract type are indicated by a function call in which a compound name for the function is used:

<type-name> $ <operator-name> (<parameters>)

The first part of the compound name identifies the type which the operation acts upon while the second component identifies the operation. Function calls of this type will always have at least one parameter: an object of data type <type-name>; for example

stack$empty(s)

token$is_op(t)

and so forth.

A brief description of the logic of Polish_gen can now be given. Polish-gen uses scan to obtain a symbol of the grammar from the input string. Scan returns the symbol in the form of a token -- a type introduced to provide efficient execution without revealing information about how the grammar represents symbols. Polish_gen then obtains the precedence relation holding between the newly scanned symbol and the symbol on the top of the stack. Finally, the action appropriate for the precedence relation is performed.

The scan procedure obtains characters from the input file via the operations infile$get and infile$putback. It makes use of the data types char and string, and operations on objects of these types. Although these types are shown as supported by the language, they could easily have been abstract types instead. In that case, the primitive predicates letter and number, for example, would have been expressed as char$letter and char$number. Only the syntax changes; the meaning of the types is the same in either case.

To sum up, Polish-gen makes use of five data abstractions: infile, outfile, grammar, token and stack, plus one purely functional abstraction: scan. The power of the data abstractions may be illustrated by considering parameters input and output. Polish-gen is completely shielded from any physical facts concerning its input and output. For example, it does not know what output device is being used, whether there even is one, and when the I/O actually takes

place, nor does it know how characters are represented there. What it does know about output is just enough for its needs: How to add a string of characters and how to signify that the output is complete. Its knowledge consists of the names of the operations which provide these services.

Defining Abstract Data Types

In the previous section, we discussed how to make use of abstract data types and how the compiler assumed when it encountered an abstract type that the meaning of that type would be provided by the compilation of another program. In this section, we describe the programming object, the function cluster, or cluster for short, whose compilation defines the meaning of a type. The function cluster embodies the idea of a data type being completely characterized by the operations on that type. A function cluster exists to support an abstract data type, and each permitted operation corresponds to a function in the cluster.

As an example, consider the abstract data type stack used by Polish_gen. A cluster supporting stacks is shown in Figure 3. This cluster defines a very general kind of stack object in which the type of the stack elements is not known in advance. The cluster parameter element_type indicates the type of element a particular stack object is to contain.

A cluster definition has two main parts: A very brief description of the interface which the cluster presents to its users,

```
stack: cluster(element_type:type, init_val:element_type)
       is push,pop,top,erasetop,empty;

   rep(elem_type:type)=(tp:integer;
                        e_type:type(elem_type);
                        stk:array[1..] of elem_type);

   push: procedure(s:rep, v:s.e_type);

      s.tp:=s.tp+1;
      s.stk[s.tp]:=v;
      return;
      end

   pop: procedure(s:rep) returns s.e_type;

      if s.tp=0 then error;
      s.tp:=s.tp-1;
      return s.stk[s.tp+1];
      end

   top: procedure(s:rep) returns s.e_type;

      return s.stk[s.tp];
      end

   erasetop: procedure(s:rep);

      if s.tp=0 then error;
      s.tp:=s.tp-1;
      return;
      end

   empty: procedure(s:rep) returns Boolean;

      return s.tp=0;
      end

   create

      s: rep(element_type);

      if exists(init_val)
         then s.stk[s.tp]:=init_val;
         else s.tp:=0;
      return s;

   end stack
```

Figure 3

and a complete definition, in the form of procedure declarations and data descriptions, of how the cluster supports this interface. Thus, the separation of what from how is clearly present in the cluster definition.

The form of the cluster interface description is:

<cluster-name> : cluster[(<cluster-parameters>)]
is <operator-list>

The <cluster-name> defines the name of the abstract type which the cluster provides. The <operator-list> lists all the operations which can be performed on objects of that type; the use of the reserved word is underlines the idea of a data type being equivalent to a group of operations. Thus, the stack cluster defines stacks as equivalent to five stack operations used by Polish_gen. The optional <cluster-parameters> defines the information which must be made available when objects of that abstract type are created. For example, a stack object is created to hold elements of a specified type (the parameter element_type); as a convenience, the object will also be initialized to contain a single element of type element_type (the parameter init_val).

The remainder of the cluster definition, describing how the abstract type is actually supported, contains three pieces of information: A description of how objects of the abstract type are actually represented; a body of code to be executed when objects of that type are created; and a number of procedure definitions.

Object Representation. Users of the abstract data type supported by a cluster view objects of that type as indivisible,

non-decomposable things. Inside the cluster, however, objects are viewed as decomposable into elements of more primitive type. The rep description

rep[(<rep-parameters>)] = <type-definition>

defines the way objects are viewed within the cluster. The <type-definition> defines a template which permits objects of that type to be built and decomposed. In general, it will make use of the data structuring methods provided by the language; data may be structured using either (possibly unbounded) arrays or PASCAL records. Both structuring methods are used in the stack cluster: A stack is represented by a record of three components, tp, stk, and e-type. The storage for the stack is in the array named stk which contains elements of type e-type, and tp holds the index of the topmost element in the stack.

The optional <rep-parameters> provide flexibility in the creation of objects. In the example, the argument specifies the type of stack element; other frequently occurring uses for the <rep-parameters> include specifying the bounds on an array or the initial value of some part of the rep.

Object Creation. The reserved word create marks the <create-code>, the code to be executed when an object of the abstract type is to be created. The cluster may be viewed as a procedure whose procedure body is the <create-code>. When a user declares a variable to be of abstract type, for example,

```
s: stack(token, grammar$eof(g))
```

one thing that happens (at execution time) is a call on the

cluster-procedure. This causes the <create-code> to be executed. The <create-code> makes use of the parameters given as part of the declaration. These parameters are in fact local to the <create-code>. When the <create-code> returns, they will cease to be defined and, therefore, may not be accessed freely in other parts of the cluster-definition (including the rep description--see below).

The code shown in the stack cluster is typical of <create-code>. First, an object of type rep[(rep-parameters)] is created: Space is allocated to hold the object as defined by the rep-description. Then, (optionally), some initial values are stored in the object. The language provides a primitive, exists, which can be used to test whether an optional parameter is present and then take appropriate action; the use of this primitive is illustrated in the <create-code>. Finally, the object is "returned" to the caller; that is, something is returned which, when passed as a parameter to a function in the cluster, will permit the object to be accessed. (What this thing is will be discussed in a later section.)

Functions. The functions in the cluster are defined by means of procedure definitions. Those functions whose names appear in the <operator-list> constitute the permissible operations on the data type. Other functions may also be defined, but they provide only local subroutines for use within the cluster and may not be called externally. The definitions of cluster functions are just like ordinary procedure definitions except in their use of type rep.

Cluster functions always have at least one parameter -- of type

rep. Because the cluster may simultaneously support many objects, this parameter tells the function the particular object on which to operate. Note that there will be a change of type with respect to this parameter between the caller and the cluster function. The compiler allows only this single case of a difference in type of parameter between calling and called procedure; the way in which this is accomplished will be described later in the paper.

In the course of its computation, the function may need to create a new object of the abstract type supported by the cluster. It may only do this by declaring the object to be of type rep -- it is not permitted to use its own type abstractly. This restriction eliminates one kind of recursion. In creating this object, the function will specify <rep-parameters> if they are required. In fact, rep was defined to take parameters in order to permit functions to create objects of type rep.

A Further Example of Clusters

As a further example, we present the cluster for grammar in Figure 4. This example actually contains two clusters: grammar and token. The token cluster is embedded in the grammar cluster. This means that it has the same access to the way grammars are represented as do the grammar cluster functions.

Why embed the token cluster in the grammar cluster? In fact, why define tokens at all? Polish_gen could have been written to accept strings from scan, to store strings on the stack, and to pass strings

```
grammar: cluster("grammar description") is eof_token,token,prec_rel;

   rep(n:integer)=(prec_table: array[1..n,1..n] of char;
                   op_symbols: array[1..n] of string;
                   maxsymbs: integer(n));

   eof_token: procedure(g: rep) returns token;
      return token(g,"eof");
      end

   prec_rel: procedure(left,right:token, g:rep) returns char;
      if ¬(token$is_op(left) and token$is_op(right)) then error;
      return g.prec_table[token$index_is(left),token$index_is(right)];
      end

   token: cluster(g:rep, s:string) is is_op,index_is,symbol;

      rep=(op:integer(0); symb:string(null); gram:*rep);

      is_op: procedure(t:rep) returns Boolean;
         return t.op>0;
         end

      index_is: procedure(t:rep) returns integer;
         if t.op>0
            then return t.op;
            else error;
         end

      symbol: procedure(t:rep) returns string;
         if t.op>0
            then return t.gram.op_symbols[t.op];
            else return t.symb;
         end

      create
         t: rep;
         i: integer(1);
         found: Boolean(false);

         t.gram:=g;
         if exists(s) then
            begin
               while i≤g.maxsymbs and ¬found do
                  if s=g.op_symbol[i]
                     then found:=true;
                     else i:=i+1;
               end;
               if i≤g.maxsymbs
                  then t.op:=i;
                  else t.symb:=s;
            end
         return t;
      end token

   create
      "build the prec_table from the grammer description"
   end grammer
```

Figure 4

to grammar$prec_rel. Although this would require one less abstract type it would be far less efficient and only apparently simpler. The problem is that a string is looked up in the table of reserved words for the grammar (op_symbols) more than once. The first search is made to determine if a string is an operator symbol; all subsequent searches are made to find the index to be used to access the precedence matrix. Because the index could have been easily obtained in the first search, and because searching is a costly operation, this is inefficient.

Still, why not have the first search return an integer which is the index of the operator or zero if the string is not an operator? Besides the fact that using an integer in that manner is bad coding practice, it also exposes information about how the grammar is represented which is of no concern to the user of the grammar cluster. (See Parnas(2) for a discussion of information distribution.) Exposing the information means that the form in which the grammar is represented is constrained because a user of the grammar cluster may depend upon testing the integer to separate identifiers and operators. It also means that the grammar cluster cannot assume the integer passed to it as an argument will always be a correct precedence table index; for example, it would have to check that the integer is not greater than the number of symbols in the table.

The way to solve these problems is to make the result of the search for reserved words a data type, token, which encapsulates the integer map between op_symbols and the precedence matrix. By

embedding the cluster for tokens in the grammar cluster, the token functions can have access to the representation of the grammar and the proper use of an integer which indexes the array op_symbols can be guaranteed.

Note that the field, op, of the rep for token is used to distinguish identifiers from operators (t.op>0 if the token is an operator token). This is the coding trick which was criticized earlier. The difference in this case is that knowledge of the use of the coding trick is limited to the token cluster. Outside the cluster the boolean operator is_op is used to distinguish the two kinds of tokens. Since the knowledge about the coding trick is contained with the cluster, the confusion it introduces is manageable.

A token is created from two parameters: the symbol string which the token is to represent, and the grammar whose reserved word table is to be searched. The use of rep in the heading of token is treated the same way as the use of rep in any other function in the grammar cluster, and therefore refers to the rep of grammar. Token also has its own rep; so, inside the body of token, these two reps must be distinguished. *rep is used to refer to the enclosing rep.

Because the index used by prec_rel is the index of the symbol in op_symbol, the string for an operator need not be stored in the rep of token but can be retrieved from op_symbol when it is needed. The token function, symbol, which does the retrieval, requires access to the grammar to do it. Since the parameters of the <create-code> of token are not accessible to symbol, it must obtain this access through

its rep parameter, t. The <create-code> for token stores a reference to g in the rep especially for this purpose. Although the use of the rep of grammar through the rep of token leads to doubtful efficiency in the example, it illustrates an important technique for sharing information.

Relationship to Previous Work

Much work has been done in the area of creating suitable mechanisms for defining data types. There is no hope of surveying all that work here, nor is it all relevant to this paper. In this section we outline the areas of work that are most closely related to clusters in that they provide in some way tools for defining abstract data types, and we attempt to characterize how the cluster approach differs from that work. The related work can be loosely divided into three categories: extensible languages, implementation specifications for a set of standard abstract operators, and SIMULA67 class definitions.

Extensible Languages

Much of the work and much of the success with extensible languages(3) has been in the area of data type definition. This work, however, has been primarily oriented toward constructive rather than abstract definitions. New types, or modes as they are frequently called, are created by constructing a representation in terms of existing types using the primitive type construction facilities of the language. Type construction facilities provided by an extensible language typically include mechanisms for defining pointers to

objects, or defining unions of distinct type classes, and for constructing aggregates of objects. These correspond closely to the facilities used here to define reps. The operations applicable to such constructive types are derived automatically from the form of the type construction rather than being explicitly defined by the creator of that type. Although, in some cases, it is possible to augment the constructively defined operations with some derived operations, it is usually the case that any tailoring of operations to match an abstract type is left to the syntax extension mechanism.

The main problem with extensible languages is that they do not encourage the use of data abstractions; instead, they tend to make them difficult to define. Because the abstract operations are normally defined syntactically while the data is defined by a construction, a user must learn two different mechanisms, and the definition, instead of being collected in one place, is split into distinct parts. Furthermore, when syntax macros are used, it is difficult to restrict access to the representation to just the functions defined for the abstract data type.

Standard Abstract Operations

The work derived from the earlier work of Mealy(4) and Balzer(5) is much closer in spirit to the approach taken here. Mealy established the view that a data collection is a map from a set of selectors to a set of values, and that operations on data collections are either transformations on the map or uses of the map to access elements. This view has led to attempts to standardize a set of abstract

operators for data collections. For example, Balzer proposed a particular abstraction for such collections which define a set of four abstract operators to create, access, modify, and destroy abstract data collections. The user would define a particular collection by specifying how each abstract operation was to be implemented. This work has been extended (e.g., Early(6)), but its primary emphasis has remained on defining a standard set of abstract operations. More complex operations are defined as procedures written in terms of these abstract operations.

Although it is useful to distinguish some abstract operations, such as "create," which have a high probability of being applicable to every abstract data type, it seems unreasonable to expect that a predetermined set of operations will suffice to manipulate every abstract data object. Therefore, leaving the selection of the operations to the creator of the type, as is done with function clusters, provides a more closely tailored abstraction.

SIMULA Classes

The language which most closely resembles, in form, the language presented here is SIMULA67(7). SIMULA class definitions have many similarities with cluster definitions. There is, however, a very important philosophical difference in these two languages which leads to several important linguistic differences. The classes of SIMULA were designed to represent and provide full accessibility to data objects. Every attribute and function in a class is accessible in the block in which the class definition is embedded. Therefore, the

actual form of the representation is always known to the user.

In contrast to this, the rep of a cluster is not accessible outside the cluster. Functions in the cluster provide the only way to access the contents of the rep and, even then, only a subset of the functions defined in the cluster may be externally accessible. As a result of this philosophical difference, the mechanisms for referencing data, the use of non-local variable references, and the use of blocks and block structuring is quite different in the two languages.

Implementation Considerations

Most aspects of the implementation of clusters will be handled in a conventional manner. There are, however, several aspects of the implementation which deserve special mention because they are non-standard or have a significant impact on the practicality of using clusters to represent abstract data.

Modules and Module-Names

The compiler accepts a module as input. A module will usually be a cluster, but will sometimes be a procedure like Polish_gen or scan. In the course of compiling a module, free variables will be encountered either as abstract data types or procedures. Note that operations on abstract data type objects are not referenced freely, since they are prefixed by the <type-name> in every case. Thus the class of free variables corresponds naturally to the modules and each

free variable constitutes a module-name.

When the compiler processes a module it builds a description-unit containing information about the module. Information held in the description-unit includes:

1. The location of the object code generated by the compiler.

2. A description of the interface which the module makes available to its users. In particular, complete information about types of all parameters and values expected by the module is maintained. If the module is a cluster, information will be kept for each function in the cluster.

3. A list of all users of the module.

Obviously much more information can be stored in the description-unit. For example, debugging information in the form of symbol tables, etc., documentation information, specification information in the form of predicate calculus descriptions of input/output relationships, and even an analysis of the rationale for the decisions made in designing the module.

We expect the compiler to provide a meaning for each module-name in the form of a map from module-name to description-unit, and from there to the module itself. The compiler obtains access to description-units by means of a multi-level library system. The library consists of a tree of directories which are structured much

like the directories in the MULTICS file system(8). Each directory represents a related collection of modules.

The library system thus provides the information necessary to map module-names into modules. However, using the entire library to map a module-name provides too much flexibility and leads to the possibility of name conflicts. Instead the compiler interprets module-names using a directory supplied by the user and possibly built specially for the compilation or group of compilations. The compiler also must know the library name of the module being compiled so the library can be updated to accomodate future maps.

Type-Checking and Object Representation

We expect the compiler to detect all errors it possibly can at compile time. A particularly important class of errors results from mismatches of types of parameters and values across module boundaries. The compiler makes use of the type information stored in the description-unit to detect such errors. Once the compiler has obtained the description-unit by using the library to interpret the module-name, it will check to be sure that the types of parameters passed correspond exactly to the expected types, and likewise for the values returned.

The only place where a non-match is permitted is when an object has abstract type in the caller and type rep in the called procedure. One of the most important aspects of the cluster representation of an abstract data type is that the actual rep of the data type is

inaccessible outside the functions which make up a cluster. Thus, it is extremely important that the change to type rep is a legitimate one. It may be that the compiler can do sufficient checking at compile time in this case. However, the technique to be described in the following paragraphs provides greater flexibility.

Morris(9) has described a technique for controlling access to data objects which can be adapted to controlling the accessibility of reps. The idea is to attach to each rep instance a value which uniquely identifies the cluster which created that instance. The purpose of this attachment, which is called a tag, is to limit direct use of the rep to that code which knows the tag, namely, the functions defined in the cluster. The tagging must be done in a way which insures that no program unit outside the cluster can learn the unique value and thereby be able to counterfeit instances of the abstract type.

In the language discussed here, unlike that used by Morris, neither the tag nor the tag checking and insertion operations are visible to the user. Whenever a cluster is compiled, a unique tag value is obtained from the operating system by the compiler. This tag value is used by the compiler to generate the code to create instances of the abstract data type and the code within the functions of the cluster which access the rep directly. Whenever an instance of the rep is created, code is generated to tag it with the cluster tag. Within each procedure which has one or more arguments of type rep, code is generated to verify the correctness of the tag on the actual arguments and, if the tags match, to allow direct access by

temporarily negating the effect of the tag. When the function returns, the access limiting effect of the tag is restored.

This approach can be used even when the code for the functions of the cluster is expanded inline, provided the operating system prevents the user from accessing or modifying compiled code without using the compiler. If the user is prevented from accessing his compiled code directly, the tag and the primitives which control the effect of the tag can be inserted inline with the rest of the function. This allows normal optimization to take place without exposing the representation.

Top-down Programming

In top-down programming, module-names will frequently be used before the modules they refer to are defined. Nevertheless, we wish to compile meanings for such modules and still have the compiler detect errors in use of types.

When a module is referenced before it is defined, a description-unit will be created for it, and type information collected on the basis of what the user-module expects. The user-module will be completely compiled and may even be debugged using a simulation of the not-yet-defined module.

When the module is finally compiled, its type information will be compared with the type information in the description-unit. In the course of this comparison, errors may be detected, and these are more likely to be errors in the user-module than in the module currently

being compiled. In order to cope with such errors, a list of all users of a module is maintained in the module description-unit. This enables the compiler to notify the programmer of the identity of the modules which are in error.

Efficiency

We believe it is helpful to associate two structures with a program: its logical structure and its physical structure. The primary business of a programmer is to build a program with a good logical structure -- one which is understandable and leads to ease in modification and maintenance(10). However a good logical structure does not necessarily imply a good physical structure -- one which is efficient to execute. In fact, the techniques employed to achieve good logical structure (hierarchy, access to data only through functions, etc.) in many cases seem to imply bad physical structure.

We believe it is the business of the compiler to map good logical structure into good physical structure. The fact that the two structures may diverge is acceptable provided that the compiler is verified, and that all programming tools (for example, the debugging aids) are defined to hide the divergence.

The language is intended to be compiled by an optimizing compiler which achieves a good physical structure in the output code. An important efficiency can be obtained from the fact that the language is flexible with respect to the meaning of an operator-use. Each operator-use may be replaced either by a call upon the corresponding

function in the cluster or by inline code for the corresponding function. Two aspects of the language design make this flexibility possible:

1. Because the syntax for an operator-use is identical in both cases, it is possible to change the compiling technique that is used without rewriting the procedure in which the operator is used.

2. The invariant portion of the cluster, the code for the functions, has been carefully separated from the rep, which holds the object dependent information; thus, inline insertion of the code is possible.

Inline insertion of the code for a function allows that code to be subject to the optimization transformations available in the compiler. Optimizing transformations, such as compile-time evaluation and common subexpression elimination, remove redundant computations, thereby decreasing the time needed to execute the operation. For example, all error checks in the stack cluster functions could be eliminated if those functions were inserted inline in Polish_gen. These standard optimization techniques should be extremely effective because the compiler is dealing with a structured program; the lack of free variables, and of goto's and other confusing control structures implies that a thorough data and control flow analysis can be performed. In other words, the compiler can benefit from the good logical structure of the program to obtain a thorough understanding of it, just like a person can.

The price paid to obtain this execution time optimization is an increase in the cost of redefining or modifying a module. (We are concerned here especially with modifications which perserve the interface; for example, using hashing to find reserved words in the token cluster instead of a linear search.) Each such modification may require the recompilation of the modules which use the modified functions inline. Since the decision to use inline code can be delayed until performance measurements indicate which sections of a system are critical, one need relinquish the flexibility of easy program modification only where a positive performance benefit would result from inline code. Note that the list of the users of the module, kept in the description-unit, can be used to cause automatic recompilation when changes are made.

Conclusion

This paper has described an approach to computer representation of abstraction. The approach was discussed both as a concept and as a part of a programming language, and examples were given of its use.

The basic idea is simple: an abstraction is viewed as an abstract data type which is completely characterized by the operations which may be performed on objects of that type. Nevertheless, the idea and its incorporation into a programming language appear to be new. Although it is possible to define such abstractions in existing programming languages, none of them supports this ability clearly and simply.

The rationale behind undertaking to develop the language was to make the practice of structured programming more understandable. Structured programming is based on a seemingly mysterious process of discovering useful abstractions. We felt that some of the mystery could be removed if the language were right -- that some of the mystery resulted from lack of understanding of what an abstraction was, and the proper language would resolve this issue.

We are convinced, based on our experiments in using the language, that we have succeeded in this direction. We feel, however, that the language also aids in another way. Although a language can never teach a programmer what constitutes a good program, it can guide him into thinking about the right things. The language described in this paper encourages the programmer to consciously search for abstractions, and to think very hard about their use and definition. Since we believe that proper selection of abstraction is the key to good design(10), we are very encouraged by the emphasis which the language places on it.

We believe that the approach to abstraction discussed in the paper can be usefully incorporated in many different kinds of langauges. It is unlikely that any language, no matter how high-level, contains all the abstractions which any person working in it would require. The abstraction building mechanism described in this paper would in fact be a useful feature of a very-high-level language. In addition, a language incorporating the mechanism would encourage structured programming and the production of structured programs --

worthy goals, no matter what level of language is being used.

References

(1) Wirth,N., "The Programming Language PASCAL," Acta Informatica, Vol. 1 (1971), 35-63

(2) Parnas,D.L., "Information Distribution Aspects of Design Methodology", Proceedings IFIP Congess (August 1971)

(3) Schuman,S.A., and Jorrand,P., "Definition Mechanisms in Extensible Programming Languages," AFIPS FJCC Proceedings, Vol. 37 (1970), 9-19

(4) Mealy,G., "Another Look at Data," AFIPS FJCC Proceedings, Vol. 31 (1967), 525-534

(5) Balzer,R.M., "Dataless Programming," AFIPS FJCC Procedings, Vol. 31 (1967), 557-566

(6) Early,J., "Toward an Understanding of Data Structures," Comm. ACM, Vol. 14, No. 10 (October 1971), 617-627

(7) Dahl,O.J., Myhrhaug,B., and Nygaard,K., The SIMULA 67 Common Base Language, Norwegian Computing Center, Oslo, Publication No. S-22, 1970

(8) Daley,R.C., and Neumann,P.G., "A General-Purpose File System for Secondary Storage," AFIPS FJCC Proceedings, Vol. 27 (1965), 213-229

(9) Morris,J.H.Jr., "Protection in Programming Languages," Comm. ACM, Vol. 16, No. 1 (January 1973), 15-21

(10) Liskov,B.H., "A Design Methodology for Reliable Software Systems," AFIPS FJCC Proceedings, Vol. 41 (1972), 191-199