An Introduction to CLU

CSG Memo 136
February 1976

Barbara H. Liskov

This paper was published in the ALGOL Bulletin. It supersedes CSG 112-1, *A Note on CLU.*

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract N00014-75-C-0661.

We believe the best approach to developing a methodology that will serve as a practical tool for program construction is through the design of a programming language such that the abstract problem solutions developed using the methodology are actual programs in the language. Several benefits accrue from this approach. First, since designs produced using the methodology are actual programs, the problems of mapping designs into programs do not require independent treatment. Second, completeness and precision of the language will be reflected in a methodology that is similarly complete and precise. Finally, the language provides a good vehicle for explaining the methodology to others.

Our research in the area of programming methodology led to a methodology [3] which combines structured programming with modularity. The fundamental activity taking place in structured programming is, in our opinion, the recognition of abstractions. Structured programs are developed by repeated analysis of a problem into subproblems to be solved by program modules. Each module is a program written to run on an abstract machine providing just those abstractions (data objects and operations) suitable for the problem being solved. The abstractions in this machine, if not already present in the programming language being used, are then realized by means of further modules. The result of this process is a program structure in which each element is a module developed to support an abstraction. The simplicity of this structure, and hence the understandability and provability of the structured program, is directly dependent on a wise choice of suitable abstractions.

We have studied what kinds of abstraction are useful in writing programs, and how such abstractions may be represented in programs. Two kinds of abstraction are recognized at present: abstract operations and abstract data types. Abstract operations are naturally represented by subroutines or procedures, which permit them to be used abstractly (without knowledge of details of implementation). However, a program representation for abstract data types is not so obvious; the ordinary representation, a description of the way the objects of the type will occupy storage, forces the user of the type to be aware of implementation information.

We believe that the user of an abstract data type is interested in how the type's objects behave, and that the behavior is best described in terms of a set of operations [5]. We developed a set of criteria about the way abstract data types should be handled:

1. A data type definition must include definitions of all operations applicable to objects of that type.
2. A user of an abstract data type need not know how objects of the type are represented in storage.
3. A user of an abstract data type may manipulate the objects <u>only</u> through the type's operations, and <u>not</u> through direct manipulation of the storage representation.

This last criterion ensures that the operations provide a complete description of the behavior of the type, and enhances the modifiability and provability of programs.

No existing language supports the use of abstract data types in a way which fully satisfies these criteria [6]. The language providing the closest match to abstract data types is Simula 67 [7]. A Simula class may be viewed as a type-definition, and as part of that definition, the programmer may include all the operations which make sense for the objects of the new type. Unfortunately, Simula does not constrain access to the objects to occur only through the operations, and thus violates criterion 3.

The next section contains an informal introduction to CLU, including an example of an abstract data type definition. This is followed by a discussion of the semantics of CLU. Then an extension of the type definition mechanism is described which permits classes of types to be defined. Finally, we conclude by discussing the current status of CLU.

THE STRUCTURED PROGRAMMING LANGUAGE, CLU

The principle motivation for the design of CLU is to permit the abstractions introduced during program design to be easily implemented via CLU modules. Two kinds of modules are provided by CLU: procedures, which support abstract operations, and clusters, which support abstract data types in a way which satisfies the three criteria discussed above. An abstract data type is defined to be a set of objects (values) and a set of operations. A cluster implements a type by defining a representation for the type's objects and by implementing the operations in terms of procedures which operate upon the representation.

CLU is a modular programming language/system. Each CLU module implements an abstraction; the CLU system maintains information about abstractions, the modules implementing them, and the relationships between abstractions and their implementations. This is done as follows:

1. The CLU system maintains a data base containing a description unit for each abstraction. At present, a description unit contains two main pieces of information: the input/output parameter type interface of the abstraction, and the CLU modules implementing the abstraction (note that there may be more than one module implementing a given abstraction).

2. A description unit for an abstraction may be added to the system before any module implementing the abstraction exists. All that is required is that a specification of the type requirements of the abstraction be provided. For functional abstractions, this consists of a specification of the types of all input and output parameters; for data abstractions, input and output parameter types must be specified for all operations.

3. The CLU system translates a single CLU module at a time. Whenever a module is submitted to the system for translation, the system checks that its input and output type requirements agree with those of the abstraction it implements. If the translation is successful, information about the module is added to the description unit of the abstraction being implemented.

4. Although each module is developed and submitted to the system independently, a means must be provided to permit modules to refer to each other (so that one module can make use of the abstraction implemented by another module). CLU modules refer to abstractions by means of externally defined names. These names have no meaning in themselves; instead, the CLU system must be provided with an association list identifying for each externally defined name the description unit of the abstraction which the programmer intends that name to represent.

5. CLU is a strongly-typed language, and complete type checking occurs at module translation time. A very important part of type checking (and one which is often neglected) is the checking of interfaces between modules. The CLU translator checks such interfaces completely; it is able to do so because the association list tells what abstractions are being used, and the description units for the abstractions contain complete information about their type requirements. In addition, modules are bound at translation time to the abstractions identified in the association list; this ensures that the translator's assumptions about the type requirements of the abstractions are valid at execution time.

6. The selection of a module to implement an abstraction may occur later than translation time of a module using the abstraction; any time prior to execution of the using module is satisfactory. In fact, a module using an abstraction can be translated in advance of the existence of a module implementing the abstraction, so top-down design and implementation are supported by the system.

CLU provides an ordinary selection of control structures: if-then and case are available for conditional testing, and while-do, repeat-until and for are available for iteration. A go-to statement is not available. The most important form of control is procedure invocation. A return statement is available to terminate an invocation (and possibly return some values); this statement may appear anywhere within the returning procedure (see Figure 1).

Block structure may be used within a single module, but there is no concept of nesting one module within another. Therefore, only local variables may be used in CLU modules (since there is no static way of attaching a meaning to global variables), and communication among modules takes place through input and output parameters.

Example of a Definition of an Abstract Data Type

An example of an abstract data type definition is presented to illustrate those features of CLU which are most novel. The type to be defined is that of integer sets; a reasonable group of meaningful operations for integer sets is:

create	creates an empty set
insert	inserts an integer in a set
remove	removes an integer from a set
has	tests whether a set contains a particular integer
equal	tests whether two sets are the same
similar	tests whether two sets contain the same integers
copy	copies a set

Ordinary set behavior is desired: a set does not behave as if it contains multiple copies of the same integer.

A cluster implementing integer sets is shown in Figure 1. A cluster definition consists of three parts:

1. interface description
2. object description
3. operation definitions

```
intset = cluster is create, insert, remove, has, equal, similar, copy;

    rep = array[int] ;

    create = oper( ) returns (cvt);
        r: rep := rep$create(0);
        return (r);
    end create;

    insert = oper(s: cvt, i: int);
        if search(s, i) > rep$high(s) then rep$extendh(s, i);
        return;
    end insert;

    search = oper(s: rep, i: int) returns (int);
        for j: int := rep$low(s) to rep$high(s) by 1 do
            if i = s[j] then return (j);
        return (rep$high(s) + 1);
    end search;

    remove = oper(s: cvt, i: int);
        j: int := search(s, i);
        if j > rep$high(s) then return;
        s[j] := s[rep$high(s)];
        rep$retracth(s);
        return ;
    end remove;

    has = oper(s: cvt, i: int) returns (boolean);
        return(search(s, i) ≤ rep$high(s));
    end has;

    equal = oper(s, t: cvt) returns (boolean);
        return (rep$equal(s, t));
    end equal;

    similar = oper(s, t: cvt) returns (boolean);
        if rep$size(s)~= rep$size(t) then return (false);
        for i: int := rep$low(s) to rep$high(s) by 1 do
            if search(t, s[i]) > rep$high(t) then return (false);
        return (true);
    end similar;

    copy = oper(s: cvt) returns (cvt);
        return (rep$copy(s));
    end copy;

end intset
```

Figure 1. The Intset Cluster.

The *interface description* of a cluster definition provides a very brief description of the interface which the cluster presents to its users. It consists of the name of the cluster and a list of the operations defining the type which the cluster implements: e.g.,

```
intset = cluster is create, insert remove, has, equal, similar, copy
```

The use of the reserved word *is* emphasizes the idea of a data type being equivalent to a group of operations; the group of operations following *is* is called the *is-list*.

Users of the abstract data type view objects of that type as indivisible, non-decomposable entities. Inside the cluster, however, objects are viewed as decomposable into elements of more primitive type. The *object description* defines the way objects are viewed within the cluster, be defining a template which permits objects of that type to be built and decomposed. For example, the representation chosen for integer sets is merely an array of integers:

```
rep = array[int]
```

This simple representation is possible because CLU provides a powerful kind of array of unbounded size. Although CLU arrays are primitive in the sense that they are supported by the CLU translator, they may be viewed just like any data type as a group of operations, and a description of the array operations is sufficient to provide a programmer with a thorough understanding of the array abstraction. A subset of the array operations is described in Table 1.

The object description is actually a type definition: *rep* is defined to be equal to the type specified on the right hand side of the equal sign. Whenever the word *rep* appears later in the cluster, it means this type.

The body of the cluster consists of *operation definitions*, which provide implementations of the permissible operations on the data type. An operation definition must be given for every operation named in the is-list. Operation definitions are like ordinary procedure definitions except the bodies of operations have access to the *rep* of the cluster, which permits them to decompose objects of the cluster type. Operations are not modules; they may be written only as part of a cluster.

Some operations create new objects of the cluster type; create is an example of such an operation. The first thing create does is to bring into existence a variable r of the representing type:

```
r: rep
```

It then initializes r to an object of the representing type; it creates the object by calling on a creating operation of that type:

```
rep$create(0)
```

This line is an example of an *operation call* which requires a compound name to be used to specify the operation. The first part of the name identifies the type of the operation, while the second part identifies the operation. Since *rep* has been defined to be equal to *array[int]*, the above operation call is the same as

```
array[int]$create(0)
```

1. array limits. Each array has an upper and a lower bound and a size. All array elements between the bounds are defined (have values); no array elements are defined outside the bounds. Three operations give limit information:

low(a)	returns the index of the lowest defined element, or the initially defined lower bound if the array is empty.
high(a)	returns the index of the highest defined element, or low(a) - 1 if the array is empty.
size(a)	returns high(a) - low(a) + 1.

2. growing arrays. Arrays are empty when initially created. They may grow in either direction one element at a time.

create(i)	returns a new empty array a with lower bound i; low(a) = i, high(a) = i - 1, size(a) = 0.
extendh(a, v)	a grows in the high direction by one element, and v is stored in that element; high(a) and size(a) increase by 1.
extendl(a, v)	like extendh, but growth is in low direction.

3. accessing arrays. Arrays may be accessed and updated in the usual way, but only elements between the bounds may be referenced. The index is interpreted absolutely (not relative to low(a)).

fetch(a, i)	returns the value in the ith element of a if low(a) ≤ i ≤ high(a), else error. Syntactic "sugar" is provided: fetch(a, i) may be written a[i].
store(a, i, v)	stores v in the ith element of a if low(a) ≤ i ≤ high(i), else error. Syntactic sugar is provided: store(a, i, v) may be written a[i] := v.

4. shrinking arrays. Arrays may shrink from either end, one element at a time.

retracth(a)	if a is non-empty, returns the value in high(a) and reduces high(a) and size(a) by 1, else error.
retractl(a)	like retracth, but for low end of array.

5. equality.

equal(a1, a2)	two arrays are equal if and only if they are the same identical array.
similar(a1,a2)	two arrays are similar if and only if they have the same limits, and they are element by element similar.

6. copy.

copy(a)	returns a new array having the same limits as a, and containing a copy of each element of a.

Table 1. CLU Arrays.

Thus a call on the create operation for arrays of integers is made.

Finally, create returns this object by the statement

```
                return (r)
```

However, the type of r is the representing type, while the user of intset expects an object of type intset. Therefore, the create operation must cause the type of r to change before r is passed to the user of intset. The heading of the create operation specifies that this conversion is to occur:

```
                create = oper( ) returns (cvt)
```

This line states that the create operation expects no input parameters, and returns a single value. The use of the reserved word cvt states that this return value will be of the cluster type (intset in this case), and that the value being returned should be converted to the cluster type from the rep type just before it is returned.

Other operations manipulate previously existing objects of the cluster type. For example, the insert operation inserts a given integer into a given intset:

```
                insert = oper (s: cvt, i: int)
```

Insert does not return any values, but instead modifies the contents of the intset object passed to it as a parameter. The use of the word cvt in

```
                s: cvt
```

again means that outside the intset cluster, s is an object of type intset, and that a conversion is to occur. However, in this case the conversion is from the cluster type to the rep type, so that whenever s is used inside of insert, it denotes an object of the rep type. The conversion occurs immediately after insert is entered.

The first line of insert

```
                if search(s, i) > rep$high(s) then rep$extendh(s, i)
```

illustrates the use of an internal cluster operation. The name search does not appear in the is-list and therefore search is not available for use by users of intset. Note that search expects an object of type rep as its first parameter:

```
                search = oper(s: rep, i: int) returns (int)
```

The call of search matches its type requirements because s has type rep inside insert. The operation call of search does not require a compound name, intset$search, because it is an intra-cluster call.

Uses of intset look very similar to the uses of array objects which appear in the intset operations. Variables may be declared of type intset:

```
                s: intset
```

and intset objects created and assigned to such variables:

```
                s :=intset$create( )
```

Operations may then be applied to intset objects:

```
                intset$insert(s, 3)
        if intset$has(s, 7) then  intset$remove(s, 7)
```

Also, intset objects may be passed to procedures and to operations of other clusters. In every case, the CLU translator will check that the called procedure or operation expects an intset object in the position in which s occurs; any other expectation will cause a type error to be detected, and the translation will not complete. Therefore, it is impossible for any procedure or operation to treat an intset object as anything but an intset object.

Access to the rep of an abstract object can occur only within a cluster operation in which a parameter or result is marked by the indicator cvt. This indicator specifies that the argument or result is considered to be of the cluster's abstract type outside the body of the operation, but of type rep inside the operation body. Thus intset objects can be accessed as objects of type rep only inside the bodies of operations of the intset cluster.

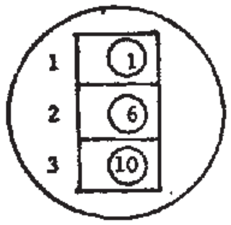
As was mentioned earlier, hiding an object's representation (criterion 3) is necessary to ensure that the behavior of the object is completely defined in terms of the type's operations. In addition, it is beneficial to software quality. Programs produced in this way are easy to modify: all changes to the implementation of a particular abstraction are guaranteed to be limited to the supporting cluster,since users of the original cluster were not able to make use of any implementation details. For example, the cluster for intset could be rewritten to store the set elements in sort order. Users of intset would be unaffected by this change (their programs would continue to run correctly) although performance differences would be noticed.

Hiding the representation is also beneficial to proofs of program correctness because it permits the proofs to be modularized along program module boundaries [8,9] .

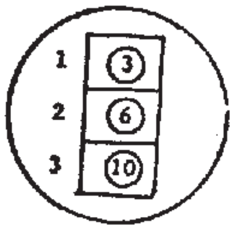
CLU SEMANTICS

The semantics of CLU is based on a sharp distinction between variables and objects. CLU object are the values which are manipulated by CLU programs. Each CLU object has a unique type associated with it. CLU objects may be simple, e.g., integer objects

or complicated, e.g., an array containing integers 1, 6, 10 in elements 1, 2, 3



However, the complicatedness or simplicity of an object cannot be observed directly; all that can be done with an object is to manipulate it using the operations defining the object's type. These operations provide the only means for making observations about objects of the type, and the operations completely define the behavior of the objects of the type. The objects of some types exhibit mutable behavior: some operations exist which will change the interior of an object without changing the object's identify. Array objects have mutable behavior; for example, the store operation, if asked to change the first element of the array above to 3, will modify the array object itself, so that at the completion of the operation the object looks like



Objects of type intset, defined in the previous section, have mutable behavior too; operations insert and remove change the state of intset objects. The objects of other types exhibit constant behavior: for such types, no operations exist to change the state of one of the type's objects. For example, integers, characters and strings have constant behavior.

CLU objects have an existence independent of particular CLU programs. They reside in the CLU universe which is like an Algol 68 heap. CLU variables, on the other hand, exist only in programs. They merely provide a convenient way for programs to reference objects. CLU provides a primitive assignment operator which permits a variable to be associated with an object: execution of

```
x := 3
```

results in the variable x denoting the object 3. CLU variables have a type, defined when the variable is declared, and an assignment is legal only when the type of the variable and the type of the object are compatible. Compatible means either the types are equal, or the variable's type is a union of several types including the object's type.

CLU follows the ordinary conventions about coercing a variable to the object it denotes whenever the variable appears anywhere but on the left hand side of :=. CLU is unusual in not viewing an array reference or record selector as a left hand side; as explained in Table 1,

```
a[i] := v
```

is merely syntactic sugar for a call on the array operation, store. The symbol a[i] is not considered to be a variable in CLU; rather it is an operation invocation.

The semantics of parameter passing in CLU is very straightforward but somewhat unusual. The identifiers of the formal input parameters defined in a procedure or operation heading are considered to be variables; thus f = proc (s: array[int], i: int)

contains the declaration of two variables s and i. When a procedure or operation is invoked, the declarations take effect, and the variables are initialized by assigning the actual parameters to them. For example, if t is an array containing 3, 6, and 10 in elements 1, 2, 3 then

f(t, 2)

is a legal call of f; it causes variables s and i to be created, and assignment:

```
s := t
i := 2
```

to be executed. The result of the call of f is illustrated in Figure 2a.

The reason that parameter passing in CLU is unusual is that assignment to the formal parameters of a procedure or operation does not affect the actual parameters. If x is an array containing 4 in element 1, and the assignment

s := x

occurs inside f, the result is that s now denotes a different object, but t is unaffected. Figure 2b illustrates the effect of s := x.

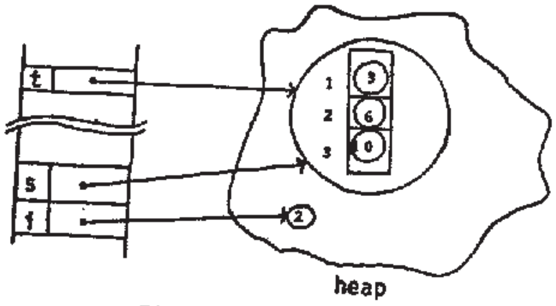


Figure 2a. Situation after f has been called.

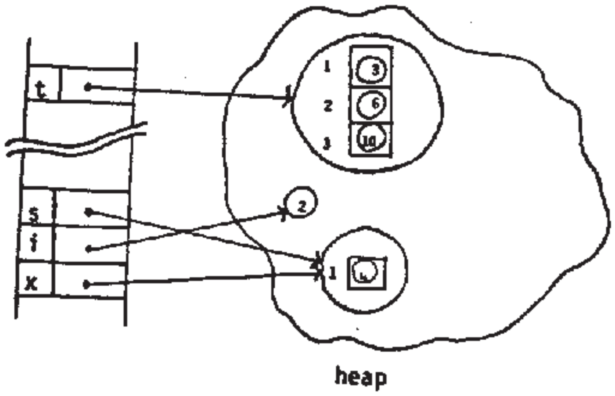


Figure 2b. Situation after s := x has been executed.

Because assignment to formal parameters inside of procedures cannot affect the actual parameters, CLU parameter passing is not call by reference, and one kind of side-effect is eliminated in CLU. We call our parameter passing call by sharing, because the object being passed is shared, as illustrated in Figure 2a. Information can be exchanged between calling and called procedures by changing the state of an object received as input; this is the only kind of side effect a CLU procedure can have.

Equality

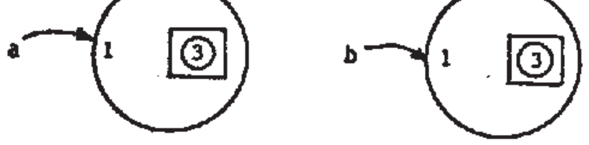
In addition to the primitive notion of assignment, a primitive notion of equality is often required in order to write meaningful programs. However, unlike assignment, which has a type-independent meaning and can be implemented automatically, equality has a type-dependent meaning. Therefore, it is not possible to provide an automatic implementation for equality. Instead, each cluster must include an equal operation (the operation which is named "equal") to provide an implementation of equality which is meaningful for the type being defined.

Although the meaning of equality is type-dependent, some general statements can be made about the meaning of equality which will help the cluster definer provide the correct definition of the equal operation. First, we can state what we expect equality to mean. Intuitively, two objects are equal if, at any time in the future, one can be substituted for the other without any resulting detectable difference in program behavior:

> Suppose T is a type, s_1 and s_2 are objects of type T, and $o_1,\ldots,o_n$ are operations of type T. If s_1 has been determined to be equal to s_2 by an application of the equal operation for T at time t, then at any time $t' > t$, any application of operation o_i, $1 \le i \le n$, to object s_1 must provide "precisely the same results" as that operation applied to s_2, where "precisely the same results" is measured by using the equal operation for the type of the object returned by o_i.

In trying to apply the above criterion when defining a type, it is helpful to distinguish between constant and mutable types. For constant types, two objects are equal if the values inside them are equal insofar as the other operations of the type are able to distinguish. For example, two complex numbers are equal if their real and imaginary components are equal; two strings are equal if they contain the same characters in the same order.

For mutable types, two objects are equal if and only if they are the same identical object. The equal operations for intset (Figure 1) and for arrays (Table 1) are examples of such operations. The necessity for such a stringent definition arises directly from the requirement, given above, that one of two equal objects can be freely substituted for the other with the same results. Suppose, for example, that the two distinct array objects, denoted by variables a and b



were considered to be equal. Now consider the program text:

```
array[int]$extendh(a, 4);
i := array[int]$size(a)
```

where i is some integer variable. Since b can be freely substituted for a with no detectable difference in program behavior, the following text should have the same behavior

```
array[int]$extendh(a, 4);
i := array[int]$size(b)
```

Clearly there is a difference in behavior; the value of i in the first case is 2 and in the second case, 1. The difference arises because, for mutable types, operations exist which change the state of objects.

Since the equal operation is present in almost every type, and its use is very widespread, CLU provides a short form for calling it. The expression

```
x = y
```

is valid only if x and y are objects of the same type, and if they are, it means

```
typeofx$equal(x, y)
```

For example, in the search operation of intset, the expression

```
i = s[j]
```

means

```
int$equal(i, s[j])
```

Since the meaning of equality is so constrained for mutable types, it is useful to have other concepts of equivalence supported by other cluster operations. One such definition is associated with the operation name "similar": two objects are similar if their contents are similar, insofar as the other operations of the type are able to distinguish. Thus, for a and b above,

```
array[int]$similar(a, b) = true
```

Another example is the similar operation of intset (Figure 1); two intset objects are similar if they contain the same integers, regardless of the order in which the integers are stored. Note that for both constant and mutable types, equality of objects implies similarity. The definer of a cluster has no obligation to provide a "similar" operation.

Copying

Often a user does not wish to have two variables share an object, or to share an object with a procedure he calls. Sharing of objects between two variables is dangerous because a change to the object through one of the variables affects the other variable. For example, starting from the situation in Figure 2b, if

```
s[1] := 5
```

is executed, the result is that x[1] will now return 5. (Recall that s[1] := 5 is syntactic sugar for the invocation of array[int]$store(s, 1, 5), and x[1] is syntactic sugar for the invocation of array[int]$fetch(x, 1).) Copying objects is much safer than sharing because such anomolies do not arise. However, the meaning of copy is not defined by the CLU semantics; instead, copy (like equal) is an operation which must be defined for each abstraction by giving an operation definition in the cluster. The reason for this is that the meaning of copy may be abstraction dependent; in fact, some abstractions may not even have a copy operation. Since copying is frequently desired, definers of clusters are urged (but not required) to provide a copy operation.

A general guideline for the definition of the copy operation, along the lines of the one given for equality in the preceding section, is:

1. for constant types,

```
        y := typeofx$copy(x)
implies
        x = y
```

2. for mutable types

```
        y := typeofx$copy(x)
implies
        typeofx$similar(x, y)
```

Examples of copy definitions satisfying the above guidelines are given for arrays (Table 1) and for the intset cluster (Figure 1).

TYPE-GENERATORS

The integer set example described earlier does not capture the concept of a set as a general receptacle for values; it defines only one particular kind of set -- a set containing integers. The concept of a generalized set presents a more powerful abstraction, the concept of "setness", than does the concept of integer set. Since the purpose of CLU is to support the use and definition of abstractions, particularly abstractions involving data, we felt it was important that CLU be powerful enough to permit a generalized set abstraction to be defined. The CLU mechanism which supports the programming of such abstractions is called a type-generator.

Type-generators differ from ordinary clusters in that they define a whole class of types, rather than a single type. Conventional programming languages contain one or more built-in type-generators. An example of such a type-generator is the array. An array defines an access mechanism which is independent of the type of data which is stored in the array. Whenever an array is to be used, the program must specify what type of data the array is to contain; e.g.,

```
        array[int]
        array[string]
```

Type definitions like these can be viewed as selecting a particular array-type from the class of such types which the array type-generator defines.

CLU permits the programming of clusters which define type-generators rather than types. An example of the set type-generator is shown in Figure 3. The set cluster is very similar to the intset cluster shown in Figure 1. The two clusters differ only in that the set cluster makes use of a type parameter to define the type of element in the set, and everywhere the intset cluster used the type int to define the type of set element, the set cluster uses the type parameter.

```
set = cluster[etype: type] is create, insert, remove, has, equal, similar, copy;

    rep = array[etype];

    create = oper( ) returns (cvt);
        return (rep$create(0));
    end create;

    insert = oper(s: cvt, i: etype);
        if search(s, i) >  rep$high(s) then rep$extendh(s, i);
        return;
    end insert;

    search = oper(s: rep, i: etype) returns (int);
        for j: int := rep$low(s) to rep$high(s) by 1 do
            if etype$equal(i, s[j]) then return (j);
        return (rep$high(s) + 1);
    end search;
      .
      .
      .
end set
```

Figure 3. The Set Cluster.

The interface description for set identifies it as a type-generator by the presence of the cluster parameter

```
set = cluster[etype: type] is create, insert, remove,
                              has, equal, similar, copy
```

All clusters defining type-generators take one or more cluster parameters. The rep for set is now

```
rep = array[etype]
```

The rep still makes use of the array type-generator, but it selects the particular array-type using the type parameter of the cluster.

In addition to appearing in the cluster interface definition and in the rep definition, the cluster parameter is also used to define the types of input and output parameters of operations; for example

```
insert = oper(s: cvt, i: etype)
```

Finally, the set cluster makes use of some of the etype operations. For example in the search operation, the equal operation of etype is used:

```
etype$equal(i, s[j])
```

A user-defined type-generator defines a whole class of types just as the built-in type-generator array does, and the rules for using type-generators are the same in either case. First it is necessary to state precisely what type is desired. This is done by using a type definition in which values are specified for the cluster parameters of the type-generator; for example:

```
intset = set[int]
newset = set[set[int]]
```

As with primitive type-generators, such definitions can be viewed as selecting particular set-types from the class of types defined by the set type-generator.

Once a type has been defined, it can be used to declare variables and make operation calls, e.g.,

```
s: intset :=  intset$create( );
t: intset :=  intset$copy(s);
ss: newset := newset$create( );
        :
newset$insert(ss, s)
```

STATUS REPORT

A preliminary version of CLU has been defined; it is described in [10]. This version permits cluster definitions in support of types and type-generators, as described above; type-generator parameters may be integer and string constants in addition to types. The built-in types include records and discriminated (tagged) unions in addition to arrays, integers, booleans, characters, and strings. A special control structure, the tagcase, is provided for manipulation of discriminated union objects to ensure that all type errors can be detected at translation time. Finally, a signal control structure is available for the reporting and handling of errors and exceptions. An experimental implementation of this first version of CLU is running on a DEC PDP-10 computer.

At present we are engaged in designing an extended version of CLU. Some of the topics under investigation include: parallelism, control abstractions, the meaning of type parameters, and polymorphic functional abstractions.

ACKNOWLEDGEMENTS

Several people have made important contributions to the design and implementation of CLU, and the author gratefully acknowledges their help. The contributions of Russ Atkinson, Craig Schaffert, and Alan Snyder have been particularly significant.

REFERENCES

1. E.W. Dijkstra. Notes on structured programming. Structured Programming, A.P.I.C. Studies in Data Processing No. 8, Academic Press, New York 1972, 1-81.
2. D.L. Parnas. Information distribution aspects of design methodology. Proceedings of the IFIP Congress, August 1971.
3. B. Liskov. A design methodology for reliable software systems. Proceedings of the AFIPS 41 (1972), 191-199.

4. P. Henderson and R. Snowden. An experiment in structured programming. Bit 12 (1972), 38-53.

5. B. Liskov and S. Zilles. Programming with abstract data types. Proceedings of ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices 9, 4 (April 1974), 50-59.

6. J.M. Aiello. An Investigation of Current Language Support for the Data Requirements of Structured Programming. Technical Memo TM-51, Project MAC, M.I.T., Cambridge, Mass., September 1974.

7. O.J. Dahl, B. Myhrhaug, and K. Nygaard. The SIMULA 67 Common Base Language. Publication S-22, Norwegian Computing Center, Oslo, 1970.

8. B. Liskov. Data types and program correctness. Proceedings of the AFIPS 44 (1975), 285-286.

9. B. Liskov and S. Zilles. Specification techniques for data abstractions. IEEE Transactions on Software Engineering SE-1, 1 (1975), 7-19.

10. C. Schaffert, A. Snyder, and R. Atkinson. The CLU Reference Manual. Computation Structures Group memorandum, Project MAC, M.I.T., Cambridge, Mass., September 1975.