24

Abset, a Programming Language Based on Sets: Motivation and Examples

E. W. Elcock, J. M. Foster, P. M. D. Gray, J. J. McGregor and A. M. Murray

Computer Research Group
University of Aberdeen

Abstract

The overall design aim of ABSET was to devise an interactive programming language in which it is possible, at will, to take or defer decisions about a program: we therefore require that decisions which are logically separable can indeed be taken separately and in any order. Further, we think it important that the language should allow the nature of each decision to be clear, particularly choice of representations.

This paper deals mainly with the treatment of declarative material in ABSET. The paper is in two parts. Part 1 presents the motivation for the fundamental ideas and their integration into the total design. Part 2 presents examples of the interactive use of ABSET, some simply to give an idea of the complete range of the declarative part of ABSET, others explicitly chosen to indicate a particular contribution of each of the fundamental ideas to the achievement of the design aims.

A large part of ABSET has been implemented and all the major examples run, though with some minor syntactic differences from the versions given here.

INTRODUCTION

In designing ABSET we have tried to solve what seemed to us to be the fundamental problems of programming. We felt that it was the lack of a clear formal organization of the very basis of programming and the bad style resulting from this that makes the management of large projects so difficult. ABSET is therefore not specially adapted to list processing, or to matrix multiplication or text processing, and so on, though such features could be introduced in it. We have concentrated on more primitive ideas: such notions as repetition, functions, the different kinds of equality, representations, the

distinction between an ordering of decisions and an order of evaluation, and the manipulation of partly-evaluated program. The treatment of the first four of these notions is founded on sets and it is from this that ABSET takes its name.

Our ideas here have grown out of earlier work reported in this Workshop series, and it is appropriate, as part of the motivation of ABSET, to sketch this earlier work.

Murray and Elcock (1968) describe an investigation of automatic improvement of a board-game (Go-Moku) playing program. The improvement was by program generation of generalized descriptions of winning board situations from program analysis of particular instances encountered in play against a human opponent. This work emphasized the need for more general solutions to problems of description and recognition of complex but well-structured data.

At this time collaboration with J. M. Foster led to a considerable widening of the original interest in program description of data. It was clear that in many problems, instead of being constrained to writing sequences of imperative statements to *construct* particular data, it would be advantageous and certainly nearer mathematical practice to be able simply to assert things *about* the structure of data. In particular, if data exists which satisfies all the assertions made about them, then these data are independent of the ordering of any operations which may be performed to construct them. We felt that it should not be necessary, at least at certain levels of program, to be concerned with such orderings. These ideas were first reported in Elcock (1968) and Foster (1968): they represent a first approach to the careful distinction that is made in ABSET between an ordering of decisions and an order of evaluation.

Further developments of these ideas led to the design and implementation of an on-line incremental compiler for assertions called ABSYS (standing for ABerdeen SYstem). Part of this work was reported in Foster and Elcock (1969).

ABSYS uses a completely declarative language with a primitive equality relation and the ability to compound assertions using *and* and *or* connectives. A written assertional program in ABSYS places no explicit constraints on the order in which particular evaluations might be performed. The on-line system is incremental in that whatever processing can be done on the basis of data already present is done.

As far as we are aware, Foster and Elcock (1969) was the first published account of an implemented general-purpose interactive language to explore this lack of concern with control. Later papers by Nevins (1970) and Fikes (1970) make contributions to this area. It is particularly interesting that Nevins, with different motivation from our own, has developed a similar treatment of the elementary equality relation, particularly in its pattern-matching aspects, and of the primitive connectives *and* and *or*.

Although an interesting language and one which gave us some moments

of great pleasure, ABSYS, even accepting the experimental limitation of being totally declarative, has a number of deficiencies. Not least is an awkward treatment of the *and* and *or* connectives which inhibited, among other things, good design of set operations in the language.

At the same Workshop, Machine Intelligence 4, at which the paper on ABSYS was presented, Robinson (1969) gave his paper on the mechanization of higher-order logic. We must acknowledge the influence of this paper on our subsequent work. It confirmed our dissatisfaction with our treatment of propositional connectives in ABSYS and was a contributory stimulus to a re-examination of the primitive notions of ABSYS.

As already mentioned, this re-examination led to a further broadening of our language objectives and to the focusing of our attention on the elementary notions of value, repetition, equality, representation, the distinction between ordering of decisions and an order of evaluation, and, not least, the manipulation of partly-evaluated program.

THE PHILOSOPHY BEHIND ABSET

We describe below some of the things we have borne in mind in producing ABSET. Two themes run through them: that it should be clear what a program says, and that the language should not force the programmer to commit himself to decisions he would prefer to postpone.

We do not suggest that ABSET is perfect in these respects.

A simple structure for the language

It is important to be able to describe the language in as simple a way as possible, not only for the purpose of teaching it, but also because this tends to simplify the implementation. We have tried to make it possible to explain the language as follows.

First, we introduce a few fundamental ideas, such as the purpose of a compiler, the nature of a function and the idea of evaluation.

Second, we say that syntax is of secondary importance. There is, of course, a primitive syntax but, beyond this, every syntactic construction for an expression can be described in terms of the application of named available functions to parameters.

Third, we describe the significance of these functions.

The grouping of decisions in a program

We would like to be able to choose any structure for a program: that is, we would like to be able to group the microscopic decisions of which a program consists in any way we please. We include here decisions about representations of data and functions. We particularly want to distinguish between the order of evaluation and the ordering of statements in program text. We do not want to be forced by the language to take compound decisions, though of

course we want to be able to make such constructions when convenient. We think these features are important for several reasons.

First, good program organization demands them: it should be possible to write program at different levels of abstraction. For example, it should be possible to indicate that an object is a function from integers to reals without deciding whether it is to be represented by a procedure or an array. It should be possible to indicate that an object is an array without deciding how big it is, or to indicate that it is a procedure without specifying the details of the corresponding algorithm. It should be possible to say those things before or after using the object.

As a good illustration of the difficulties created by ignoring them, consider the writing of library procedures in ALGOL 68, for which see section 0.1.44 of Wijngaarden *et al.* (1969). This can be done only if the decision about all the modes of the parameters is made at the time of definition of the library procedure, so that, for example, a general procedure to sort pairs of integers and references cannot be written because we must decide, too early, exactly what kind of objects the references are to.

Second, we need them for descriptive programming. In many cases we would be content to write merely a specification of the required constraints on our data, and leave the problem of satisfying these constraints to the system. That is, we would like to leave out the information about the order of evaluation (an extreme version of this would be to write our descriptions in the predicate calculus, but we do not feel that the strategies of current theorem provers are good enough to make this a practical approach at present).

Third, they help in making programs easier to manipulate, in compilation, in proofs of their properties, and in turning programs, or partially evaluated programs, back into text.

Fourth, they are most important for interactive use of the language, both because the order of taking decisions cannot be thought out in advance and also because program can be partly evaluated as the programmer proceeds, without waiting for him to complete his definitions.

The separation of the statement of the problem from the problem solver

All interesting programs are finite presentations of a potentially-infinite set of evaluations. The two notions of the finite program, and the possible evaluation of it by an interpreter or problem solver, are quite distinct. We wish to keep this distinction clear, both for the sake of the programmer and because we may wish to have different interpreters (we call them sequencers to emphasize a different aspect of them) in different circumstances.

To obtain this feature in full we require that the representation of a partly-evaluated program produced by a particular sequencer shall be acceptable data to other sequencers (see the section below on availability of the data structures used for evaluation).

The availability of the data structures used for evaluation

It is very convenient to be able to manipulate partly-evaluated program. So far the only significant use that we have made of this feature has been in the printing of the constraints on objects in a partly-evaluated state (see the section on generating text in ABSET and the associated examples). However, we intend to base much of our future work on this feature.

To take an example, consider a differentiation program in a conventional language. This will usually be written to take in a formula in text form, translate it to some internal form suitable for the program, carry out the differentiation, translate back into text, and print out the result. In ABSET the formula could be the result of partial evaluation and the result or intermediate results could be further evaluated at any time. The powerful features of the evaluation mechanism are always available for working 'on the side'.

Repetition and sets

We decided that the elementary ideas of sets would form the best basis for a programming language. Many things we want to do in programs fall naturally into this terminology, and we regard it as a good foundation for ways of achieving repetition. Rather than use **go to** statements or recursion, we prefer to use repetition formulae, such as 'this is true for all members of this set' and the notion of image sets. Quite apart from their clarity, such constructions have the advantage of giving rise to structures more amenable to partial evaluation and further processing.

Representations, equality, and sets

It is an important aid to clarity in a program to know what is representing what. The process of programming consists of representing certain sets of objects and the functions between them by means of 'lower level' sets and functions. Much confusion in programming languages comes from not explicitly recognizing this fact. Languages (like ALGOL 68) provide basic sets and functions which the programmer may use to represent his problem, but no declarative way of indicating the representations at any level higher than this.

We have provided three features intended to be helpful in these respects. First, we can define sets in a general way, for example, as those items satisfying a predicate (see the section on sets below). Second, we insist that the domain and range of every function is stated in terms of such sets and that functions are total, that is, defined everywhere in the domain (of course we do not insist that this information is given at any particular time in relation to the definition of the function). Third, we attempt to treat the notion of equality carefully. For example, sometimes we might use a list to represent a set. When we are considering the functions on sets we have one notion of equality but when we are thinking of the object as a list, in order to represent the set

operations, we need a different equality. We have therefore chosen a notation which brings out the fact that equality is an equivalence relation, and which allows us to have any number of such equivalences. Furthermore, we have taken the same care over the equality implied by the parameter passing mechanism (see the section on function application below).

Constructors and projectors

We prefer to use constructors in programming, rather than projectors, wherever possible and appropriate. Thus

```
A=[B, C, D]; E=[D, C, B];
```

is clearer than

```
HD(E)=HD(TL(TL(A))); HD(TL(E))=HD(TL(A));
HD(TL(TL(E)))=HD(A);
```

INFORMAL DESCRIPTION OF ABSET

In this section we follow the method of language description referred to above, that is, we explain the purpose of the compiler, the nature of functions, and the idea of evaluation, and we briefly describe part of the syntax. In this paper we do not explain in detail the primitive functions of the language, but in the section of examples some of the functions are illustrated.

Program text, the compiler, and program evaluation

A piece of program text consists of expressions and statements. Expressions are constants or objects. Objects are identifiers or functions applied to expressions (*see* below).

So

```
1
A
F(A)
A+B
A=B
```

are expressions. Note that infixed operators, and other syntactic constructions not illustrated, stand for functions applied to expressions.

Since the text of a program is much easier to manipulate on paper than any other representation of the data in the computer, we shall pretend that there is a notional interpreter of text. Statements are certain expressions of the program which we insist must denote the constant TRUE. This is done by placing a semi-colon after them which gives this information to the interpreter. It is the job of this interpreter to try to find constants to associate with the objects in such a way as to associate the statements with TRUE and to be consistent with the definition of the primitive functions.
For example,

```
A+B=3 AND A=1
```

is an expression and the interpreter is not applicable to it, but

A+B=3 AND A=1;

is a statement and is interpreted as meaning that the whole expression is associated with TRUE, therefore A+B=3 is associated with TRUE and so is A=1 by definition of AND, therefore A+B is associated with 3 and A with 1 by definition of =, and therefore B is associated with 2 by definition of +.

The classification of constants in ABSET includes primitive type sets such as integers, reals, booleans, strings, quoted identifiers, code entries, and references. Other types can be constructed as described in the section on sets.

The compiler translates text of statements into a representation in the computer and applies a representation of the interpreter (a sequencer plus the function definitions) to this to construct the values of the expressions (that is, the representations of the constants associated with them). The action of the interpreter in the current implementation of ABSET is illustrated in the section on annotated examples of ABSET programs.

Functions, their application to expressions and the idea of equality

A function is a rule for associating one value with another. A function applied to a value is an expression standing for the associated value. Thus F(A) is an expression which must be associated by the interpretation with the value belonging to A in F. The argument and result of a function are restricted to lie in certain sets, the domain and range of the function. These must be specified as part of the definition of every function.

In ABSET we insist that all functions are total, that is, they are defined everywhere in their domain. The application of a function to an argument not in its domain is an error, and when ABSET detects this it prints an error message.

In ABSET all functions are single valued, that is, all evaluations of F(X) yield equal results. This raises the interesting question of what interpretation of equality is used in this notion of single-valuedness and indeed the general question of what we mean by equality.

Equality is a concept which has caused difficulty in various programming languages (including ABSYS) and we consider that a careful treatment of it is very important. Much of the difficulty arises from having just one kind of equality: in ABSET we can have any number. By an equality we mean an equivalence relation on values which has the substitution property in some class of functions (that is, the equality of X and Y implies the equality of F(X) and F(Y) if F is one of these functions). The point of difficulty is the choice of the class of functions. If we allow ourselves only one equality then either we have substitution over the whole class of ABSET functions, or over some fixed arbitrary subset of it, neither of which is satisfactory.

Let us consider this point in more detail. The process of programming consists of representing the values and functions of our problem in terms

of the constructs available in the programming language. For example, if we are concerned with a problem in syntax analysis we first represent it in terms of concepts such as syntactic production and substitution. Then we represent these in terms of list-processing concepts such as CONS and NIL and finally these are represented by machine operations on addresses and numbers. If equality is to have the substitution property in all these functions we are in trouble. For two lists can be equal only if they are indistinguishable to all functions, including the machine ones. Likewise syntax productions, represented by lists, can be equal only if the lists are equal. This intrusion of low-level concepts (for example, of machine address) into material really concerned with lists (that is, ordered sets) leads to unnatural requirements on the programmer.

In ABSET we allow many equalities by associating them with sets. When a new set is introduced an equivalence relation is defined for it. When we write

```
A=B AS FRACTIONS;
```

we mean that A and B are equivalent according to the equivalence relation belonging to the set FRACTIONS. A default equivalence relation is used if we write

```
A=B;
```

Now whenever we define a function over a set we imply that the equivalence relation of that set has the substitution property in the function.

Let us return now to the single-valuedness of functions. Clearly this means that the evaluation of F(X) and F(Y) where X and Y are equivalent in the domain set of F yields results which are equivalent in the range set of F.

Some further comments on the equality of formal and actual parameters of compound functions is made in the section on the syntax of ABSET.

Breadth-first evaluation

Because a program is a presentation of an infinite set of possible operations of evaluation, there is a danger that if a bad order of evaluation is chosen it may be impossible to find a solution which could be found by some other order. We are not considering here the difficulties of the restricted amount of time and storage, but the possibility of the sequencer involving itself in an infinite set of operations before reaching a crucial one. That is, unless we are careful the sequencer may not be complete. To avoid this we guarantee that every operation of evaluation implied by the program will be treated within a finite number of steps.

As an example of this we can consider the enumeration of the elements of the Cartesian product of two infinite sets. We must use a Cantorean breadth-first enumeration, for example, (1 1, 2 1, 1 2, 3 1, 2 2, 1 3 . . .) and not the easier square enumeration (1 1, 1 2, 1 3, 1 4, 1 5 . . .).

The ABSET primitive sequencer

We call the unit of processing handled by the ABSET primitive sequencer a

job. A job consists of a reference (of unique type) to a data cell for the job. The data cell is a list cell of references to the objects to be processed by the machine code associated with the job. The data cell also contains a reference to a unique 'job-descriptor-cell' which contains information common to all activations of the job including the address of the entry to the machine code associated with the job.

An object referenced from the data cell of a job may be of type 'novalue' with the significance that the ABSET interpreter has not yet inferred a value for that object. An object of type 'novalue' itself contains a reference to a list of jobs (possibly the nil list). This list is extendable: we call the process of adding a job to it 'attachment' of the job to the object.

A function in ABSET is represented by a job-descriptor-cell. As well as the code entry this descriptor cell contains information such as the number of parameters of the function and the size of the data cell of the described job (the code for the job may require own workspace). This information is used by the compiler to implement function application by the creation of a job whose data cell refers to the parameters of the function and the job-descriptor-cell of the function.

In the section 'a simple structure for the language' we stressed that every syntactic construction for an expression can be described in terms of the application of named available functions to parameters.

We can now complete the description of both the compiler and of the process of evaluation. When the compiler processes text, for each expression processed the compiler applies the relevant functions to their parameters thereby creating jobs, and it also attaches each job to those of its parameters which at that moment are of type 'novalue'. Each job is added to the primitive sequencer's list of waiting jobs.

Essentially then, an object of type 'novalue' is a reference to a list of jobs each of which references that object (among others). The syntactic constructions NEW and ' are such that application of their associated functions creates new 'novalue' objects.

When the sequencer is called it removes the first job from its list of waiting jobs, places the job reference in a globally-named location, and jumps on the job entry. The code for the job is therefore written on the basis that, when the job is active, the data cell of the job is referenced from this global location. Typically the code for a job is written so that, when entered, the set of parameter references in the data cell of the job is examined. There are three possibilities:

(1) the job is unconstrained: that is, there are not enough parameters with values to allow any inferences to be made: the job terminates by effectively returning control to the primitive ABSET sequencer;

(2) the job is partly constrained: that is, there are sufficient parameters with values for the value of the remaining parameters of type 'novalue' to be inferred. The giving of these inferred values to the remaining parameters uses

a mechanism which, at the same time as giving the parameter object the inferred value, adds the list of jobs associated with the object to the sequencer's list of waiting jobs. Control is again finally returned to the sequencer;
(3) the job is totally constrained: that is, the set of parameters with values is such that it is simply a question of determining whether the constraints over them are satisfied or not. If the constraints are satisfied, control is simply returned to the primitive sequencer. If they are not satisfied, termination is by an error exit which arranges for appropriate action to be taken (for example, the setting up of jobs to take diagnostic action) before again returning control to the primitive sequencer.

A final remark: the compiler is itself a job. When active it has the good manners to 'give way' at appropriate moments: that is, it unilaterally adds itself to the (end of) the sequencer's list of waiting jobs and then returns control to the sequencer.

Some notes on the syntax of ABSET

In the section on 'a simple structure for the language' we said that syntax was of secondary importance: it should be stressed that this was in the context of explaining the language.

Good syntactic constructions can however contribute a great deal to the use of the language, to clarity and good style. We have tried to provide good constructions in ABSET.

The syntax of ABSET is mostly conventional. The constants in the various basic sets are written in the usual way (for example, 1 or TRUE or 'JOHN').

ABSET identifiers are

(1) a sequence of letters, digits and hyphens beginning with a letter;
(2) a sequence of digits;
(3) any single character from ` ' , ; [ ] £ " ( ).
(4) any sequence of characters chosen from the remaining teletype symbols.

An identifier is terminated by space or newline or by any character from another group.

Group (1) is used in ABSET for names of user objects. Group (2) identifiers are interpreted as integers. Group (3) are reserved identifiers in ABSET. Group (4) is used in ABSET for user objects: typically infix operators. Certain combinations from groups (1) and (4) have particular meanings in ABSET.

New identifiers are introduced by priming them or by a NEW statement (*see* section on 'a general demonstration of ABSET'). The textual scope of identifiers is controlled by BEGIN and END in a conventional way except that the END construction allows promotion of identifiers out to the next enclosing scope level (*see*, for example, section on 'text in ABSET: illustrated examples').

Compound functions are introduced in a way similar to the lambda-calculus. For example,

```
        INC' ISIN INTEGERS→INTEGERS;
        INC=MAP A' TO A+1;
and     INC'=MAP A': INTEGERS TO A+1: INTEGERS;
and     INC'=MAP A'−1 TO A;
        INC ISIN INTEGERS→INTEGERS;
and     INC'=MAP A': INTEGERS TO B': INTEGERS
                WHERE B=A+1;
```

all introduce equivalent functions. The primes on A and B indicate that the scope is the function. The infix WHERE insists to the interpreter that the following expression must denote TRUE.

Note that the interpretation of INC when applied is that the local objects and constraints are copied and the actual parameters asserted to be equal to the formal parameters according to the equivalence relations of the appropriate sets. Thus

```
        INC'=MAP A'−1: INTEGERS TO A: INTEGERS;
          X'=INC(Y');
```

is equivalent to

```
        NEW X', Y';
        BEGIN
            Y=A'−1 AS INTEGERS;
            X=A AS INTEGERS;
        END;
```

TEXT IN ABSET

Two main ideas underlie the conversion of partly-evaluated program into text.

First, it seemed attractive to have a single language for communication between programmer and machine and between machine and programmer: that is, it should be possible to take partly-evaluated program and re-express it as valid ABSET text, which, given a suitable state of the system, could be read as a program by this system.

Second, in an interactive programming system the user does not want to have to sit through lengthy printouts of the text representing a large structure in complete detail when he is likely to be interested only in a small part of the total structure. On the other hand, there will be times when he does want all the available information about the structure. What is required is the ability to construct text representation of structures at controllable levels of detail.

The simplest way in which the text representing a structure can be presented at controllable levels of detail is to represent substructures by identifiers which are paired with these substructures in some way.

An ABSET dictionary is a set of pairs where each pair consists of an

identifier and a value. There can be only one occurrence of an identifier in a dictionary, so a dictionary can be thought of as a mapping from identifiers to values. (Not a one–one mapping, as two different identifiers can be paired with the same value.) An environment consists of a list of dictionaries and so also represents such a mapping if the convention is adopted that the first occurrence of an identifier in a dictionary of an environment is the one that indicates which value is to be paired with that identifier in the environment. An 'inverse' mapping from values to identifiers can be obtained from an environment if, given a value paired with at least one identifier in the environment, an arbitrary choice is made from the identifiers paired with that value in the environment.

The primitive text mechanisms provided in ABSET require as input not only the object to be texted, but also an environment with respect to which the structure is to be texted. Any substructures which are in the domain of the 'inverse' mapping of the environment will be represented by their associated identifiers in the text produced.

The level of detail in the text is controlled by the extent to which the substructures of the object are named in the environment supplied to the primitive text mechanisms. Of course, in the case of simple objects, such as integers, the environment will not be used in the texting process.

Further facilities are provided whereby a structure can be scanned, prior to creating its text representation, and names invented for parts of the structure which are not named in the input environment, these names then also being used in texting the structure. This creates a text representation of the structure containing less detailed information about the structure than would otherwise have been the case. The names invented have to be paired with their new values in a fresh dictionary, and that dictionary added to the environment input to the text mechanism, so that if further detailed texting of part of the structure is required, the values paired with the names used in the text representation can be obtained from this new environment and the appropriate ones texted to a further level of detail.

A number of examples illustrating the ABSET text facility are included in the set of annotated examples.

Sets in ABSET

In this section we describe the defining properties of sets, some primitive sets, and some of the ways in which new sets can be formed. Other illustrations can be found in the examples. We refer also to the question of the definition of the domains of functions of more than one argument.

Sets are defined as the domain of the following functions.

(a) IN, which gives a function in UNIVERSE→BOOLEANS. This is the usual set-membership predicate for the set.

(b) EQINSET, which gives the equivalence relation over the set referred to previously.

(c) SITEM, which gives an item of the set unless the set is empty, in which case it gives an arbitrary item (cf. Hilbert's epsilon operator).
(d) SREM, which gives a set which is the same as the original except that it does not contain the item selected by SITEM.

Thus, for example, IN(SET1, ITEM) is a truth value. We can also write this with an infixed operator as ITEM ISIN SET1.

There are a number of (mostly infinite) primitive sets, such as INTEGERS, SETS, UNIVERSE, and BOOLEANS.

There are many functions which yield sets as results. They all work by specifying the results of the four projectors listed above and so they are not primitive, but can be defined using MAP. To assist in this we mention the function MAKESET which yields a set. Its four arguments are the results of the four projectors, except that instead of giving the result of SREM we give a function which, when applied to the value of SITEM, gives SREM (this is to avoid constructing the set explicitly). We give below the definition of one of these functions and describe some others formally.
(a) The definition of UPTOF (which has the infixed operator UPTO).

```
UPTOF=MAP LOW': INTEGERS, HIGH': INTEGERS TO
        MAKESET(BETWEEN(LOW, HIGH),
                EQF,
                LOW,-
                MAP LOW TO LOW+1 UPTO HIGH);
```

(b) The function CUPF (infixed operator CUP) gives the union of two sets.
(c) The function POWF (infixed operator→) gives the set of functions from one set to another.
(d) The function THOSEF gives the set of all items of a given set which satisfy a given predicate. There is a special syntactic construction for this related to MAP, for example,

```
THOSE X': 1 UPTO 1000 SATISFYING PRIME(X)
```

It frequently happens that the domain of a function of more than one variable is not a simple Cartesian product. Since we insist that functions should be total we may have to use slightly more complex constructions than we have illustrated before. For example, consider the function APPLY.

```
APPLY=MAP F', X' TO F(X);
```

The domain of this function is not simply the product of FUNCTIONS and UNIVERSE, since when we have chosen a function it is only applicable to items in its domain. We can solve this in two ways. The first is to write, for example,

```
APPLY'=MAP F', X' TO F(X)
APPLY ISIN (THOSE P' & Q': FUNCTIONS × UNIVERSE
        SATISFYING Q ISIN DOMAIN(P))→UNIVERSE;
```

The second is to write

```
APPLY=MAP F': FUNCTIONS TO
    (MAP X': DOMAIN(F) TO F(X): RANGE(F)): FUNCTIONS;
```

Since a function of more than one variable is exactly equivalent to a function of one variable (the first) of which the result is a function from the remaining variable to the result, we allow the following with exactly equivalent effect.

```
APPLY=MAP F': FUNCTIONS, X': DOMAIN(F)
        TO F(X): RANGE(F);
```

ILLUSTRATIVE EXAMPLES OF THE INTERACTIVE USE OF ABSET

The first of the examples below, 'a general demonstration of ABSET', is presented to give an idea of the range of the declarative part of ABSET. Although the example begins with the comment that not all the facilities demonstrated are available, it should be emphasized that the only major feature missing in the current implementation is the 'AS' construction discussed in the section on 'functions and their application to expressions'.

The remainder of the examples have been chosen to illustrate particular points made in the earlier part of the paper.

In all the examples we have indicated material typed by the programmer in upright type and material typed by the machine in italic type.

The 'end of message' symbol used by the programmer in interactive mode is a non-printing character. It can be used by the programmer *anywhere* in the text he is typing. The examples can therefore be read as if the material being typed by the programmer was being interpreted at word level.

EXAMPLES OF ABSET

We give first a general demonstration and then illustrate more specific points.

```
$ GENERAL DEMONSTRATION OF ABSET $
$ COMMENTS ARE ENCLOSED IN PAIRS OF DOLLAR SIGNS $
A'=1;
$ THE PRIME INTRODUCES THE NEW IDENTIFIER A, THE SEQUENCER IS
INSTRUCTED TO MAKE THE EXPRESSION A=1 HAVE THE VALUE TRUE $
A...      $ WE ASK FOR THE VALUE OF A TO BE TYPED $
1      $ ABSETS REPLY $
A=B';
B...
1
TRUE...      $ TRUE IS A CONSTANT $
TRUE
FALSE;      $ WE SAY THAT FALSE MUST BE TRUE $
UNSATISFIABLE ASSERTION:
EQ(FALSE, TRUE)
$ ABSET REFUSES TO DO THIS AND TYPES AN ERROR MESSAGE $
```

```
1=2;
UNSATISFIABLE ASSERTION:
EQ(1, 2)        $ ANOTHER ERROR MESSAGE $
TRUE AND FALSE...        $ ILLUSTRATING THE FUNCTION AND $
FALSE
(TRUE AND C')=FALSE;
C...
FALSE        $ ILLUSTRATING A MORE COMPLEX EVALUATION OF AND $
NOT(FALSE)...
TRUE
NOT(A')=TRUE;
A...
FALSE
(TRUE AND NOT(B'))=FALSE;
B...
TRUE
(A' AND B')=TRUE;
A...
TRUE
B...
TRUE
A'=B'; B=C'; C=D'; D=E'; E=F';
C=3;
A...
3
F...
3
A'=B'; B=C';
B'=5;
$ INTRODUCING A NEW B DOES NOT SPOIL THE RELATION BETWEEN
A AND C $
C=8;
A...
8
'QAZWSXEDCRFV'...        $ A QUOTED IDENTIFIER $
'QAZWSXEDCRFV'
"THE RAIN IN SPAIN FALLS MAINLY ON THE PLAIN"...        $ A STRING $
"THE RAIN IN SPAIN FALLS MAINLY ON THE PLAIN"
S1'="JOHN LOVES";
S2'="THE GIRL NEXT DOOR";
$ <> IS THE INFIX CONCATENATION OPERATOR $
S1<>S2..."JOHN LOVES THE GIRL NEXT DOOR"
S2<>S1..."THE GIRL NEXT DOOR JOHN LOVES"
[1, 2, 3]...        $ ABSET NOTATION FOR LISTS $
[1, 2, 3]
[A', 2]=[3, B'] AS LISTS;        $ THE SET LISTS IS KNOWN TO ABSET $
A...
3
B...
2
```

```
[Q', W']=[E', R'] AS LISTS;
Q...
Q=E
$ ILLUSTRATING THE PRINTING OF PARTLY EVALUATED PROGRAM.
THE CONSTRAINTS ON Q ARE PRINTED BECAUSE IT HAS NO VALUE YET $

1+2...      $ PLUS FOR INTEGERS $
3

1+S'=7;
S...
6

A'+B'=C';
B=D'+E';
A=1;
B...      $ MORE COMPLEX CONSTRAINTS $
SUM(1, B)=C AND SUM(D, E)=B

1 ISIN INTEGERS...
TRUE

1 ISIN INTEGERS CUP BOOLEANS...
TRUE

1 ISIN INTEGERS CAP BOOLEANS...
FALSE

INC'=MAP A': INTEGERS TO A+1 : INTEGERS;
$ A FUNCTION DEFINITION $
INC(6)...
7

INC(A')=8;
A...
7

DEC'=MAP A'+1 : INTEGERS TO A : INTEGERS;
$ NOTE THE FORM OF THE PARAMETER $
DEC(4)...
3

G'=MAP [A', B'] : LISTPAIRS(INTEGERS, INTEGERS) TO A+B+1 : INTEGERS;
$ THE SET CAN BE ANY EXPRESSION $
G([P', 3])=8;
P...
4

Z'=LISTPAIRS(LISTS, LISTS);
H'=MAP [A', B'] : Z TO [B, A] : Z;
H([[1, 2], [3, 4]])=[T', Y'];
T...
$ NOTE THAT THE SET Z CONTROLS THE EQUALITY OF THE ACTUAL AND
  FORMAL PARAMETERS $
[3, 4]

I1'=5 UPTO 20;
$ THE SET OF INTEGERS I, 5<I<20: I1 IS NOT EXPLICATED $

SOME X' : I1 SATISFIES X>10...
TRUE

ALL X' : I1 SATISFY X>10...
FALSE
```

```
I2'=18 UPTO 3000;
19 ISIN I1 CAP I2...
TRUE
288 ISIN I1 CAP I2...
FALSE

10&300 ISIN I1><I2...
$ >< DENOTES CARTESIAN PRODUCT AND & DENOTES A MEMBER OF
  A CARTESIAN PRODUCT $
TRUE
300&10 ISIN I1><I2...
FALSE

Q'=THOSE A'&B' : I1><I2 SATISFYING A+B=200 OR B-A=200;
10&300 ISIN Q...
FALSE
5&195 ISIN Q...
TRUE
7&207 ISIN Q...
TRUE

$ IMAGE SETS $

9 ISIN IMAGE 1 UPTO 4 UNDER MAP X' : INTEGERS TO X*X : INTEGERS...
TRUE

$ SET ITERATION: SIMILAR TO STRACHEY'S LIT $

SIT(1 UPTO 4, 0, SUM)...
$ I.E. SUM(1, SUM(2, SUM(3, SUM(4, 0))))... $
10

$ THE WHERE CONSTRUCTION: WHEREF(A, B) HAS THE VALUE OF A WITH
  B ASSERTED: 'WHERE' IS THE ASSOCIATED INFIX OPERATOR $
A' WHERE A=1...
1
A'=1 WHERE A=2...
FALSE

$ APART FROM SYNTACTIC CONVENIENCE IN USE OF CONSTRUCTORS AS
IN E.G. S' WHERE X=[R', S, T'] THE WHERE CONSTRUCTION PROVIDES THE
ASSERTIVE EQUIVALENT OF THE 'BODY' OF A LAMBDA EXPRESSION AS
IN THE EXAMPLE IMMEDIATELY BELOW $

$ FOR CONVENIENCE, A SYNTACTIC CONSTRUCTION IS PROVIDED FOR
CONDITIONAL EXPRESSIONS.
THUS:
IF T THEN A ELSE B $

DOG-NOT-CAT'=MAP X' : BOOLEANS TO
  (IF X THEN 'DOG' ELSE 'CAT'): QUOTED-IDENTIFIERS;
DOG-NOT-CAT(TRUE)...
'DOG'
DOG-NOT-CAT(FALSE)...
'CAT'

F'=MAP X' : BOOLEANS, Y' : INTEGERS TO
  (IF X THEN INC ELSE DEC) (Y) : INTEGERS;
F(TRUE, 6)...
7
F(FALSE, A')=5;  A...
6
```

```
$ INTRODUCTION OF NEW INFIX OPERATOR WITH CHOSEN INTEGER
LEFT AND RIGHT PRECEDENCES $
IS-FACTOR-OF'=MAKEOP(233, 234, MAP A' : INTEGERS, B' : INTEGERS TO
  (SOME I' : 1 UPTO B SATISFIES A*I=B) : BOOLEANS;)
3 IS-FACTOR-OF 9...
TRUE
4 IS-FACTOR-OF 11...
FALSE

$ END OF DEMONSTRATION $
```

An ABSET example to illustrate the action of the primitive sequencer

The example below uses an ABSET text facility (*see* examples on text below for further details) to illustrate the action of the ABSET sequencer in 'slow motion'.

```
$ ABSET SEQUENCER EXAMPLE $

NEW A', B', C', D';
A=B+C*D;
A=>       $ ANOTHER WAY OF PRINTING $
BEGIN
SUM(B, VAR1')=A
$ A HAS NO VALUE: IT IS DESCRIBED AS CONSTRAINED TO BE THE SUM
OF B AND A VARIABLE (VAR1) WHICH DOES NOT HAVE AN ASSOCIATED
IDENTIFIER OUTSIDE THE SCOPE OF "BEGIN" $
B...    SUM(B, VAR1)=A
VAR1...    MULT(C, D)=VAR1 AND SUM(B, VAR1)=A
$ VAR1 HAS NO VALUE: IT IS DESCRIBED AS SATISFYING THE GIVEN
CONSTRAINTS $
C...    MULT(C, D)=VAR1
D...    MULT(C, D)=VAR1
END;
A=3;
A=>
BEGIN
3
$ THE STATEMENT A=3 HAS BEEN PROCESSED BY THE ABSET SEQUENCER:
A VALUE (3) HAS BEEN INFERRED FOR A: THE CONSTRAINTS PREVIOUSLY
ASSOCIATED WITH A HAVE IN TURN BEEN PROCESSED BY THE ABSET
SEQUENCER BUT WITHOUT GIVING RISE TO ANY FURTHER INFERENCES
ABOUT B OR VAR1 $
B...    SUM(B, VAR1')=3
$ NOTE HOWEVER THAT THE CONSTRAINTS ON B, VAR1, C AND D ARE
CHANGED $
VAR1...    MULT(C, D)=VAR1 AND SUM(B, VAR1)=3
C...    MULT(C, D)=VAR1
D...    MULT(C, D)=VAR1
END;
B=1;
A=>
BEGIN
3
B...    1
$ THE CONSTRAINTS PREVIOUSLY ASSOCIATED WITH B HAVE NOW BEEN
PROCESSED BY THE ABSET SEQUENCER. A VALUE (2) HAS NOW BEEN
INFERRED FOR THE OBJECT PREVIOUSLY CALLED VAR1: NOTE THE NEW
FORM OF THE CONSTRAINTS ON C AND D $
```

```
C...    MULT(C, D)=2
D...    MULT(C, D)=2
END;
$ FINALLY $
C=2;
A=>
BEGIN
3
B...    1
C...    2
D...    1
$ THE CONSTRAINTS PREVIOUSLY ASSOCIATED WITH C HAVE NOW BEEN
PROCESSED: A VALUE (1) HAS BEEN INFERRED FOR D: THE CONSTRAINTS
ON D HAVE BEEN PROCESSED AND FOUND SATISFIED: END OF EXAMPLE $
$ ANOTHER, MORE COMPACT, EXAMPLE ILLUSTRATING THE ACTION OF
THE ABSET INTERPRETER $
$ A PRELIMINARY DEFINITION OF A PRINTING FUNCTION $
TELLME' : MAP B'=BOOLEANS, S' : STRINGS TO
  TYPE(S<>IF B THEN "TRUE" ELSE "FALSE") : BOOLEANS;
$ ILLUSTRATION OF TELLME $
TELLME(A', "A IS");
$ I HAVE ASKED ABSET TO TELLME IF A HAS BEEN INFERRED TRUE OR
FALSE $
A;    A IS TRUE
$ END OF ILLUSTRATION OF TELLME $
$ INTERPRETER EXAMPLE CONTINUED $
NEW A', B', C', D', E';
TELLME(A, "A IS");
TELLME(B, "B IS");
TELLME(C, "C IS");
TELLME(D, "D IS");
TELLME(E, "E IS");
A=(B AND C) WHERE C=(D OR E OR B);
NOT(B);    B IS FALSE    A IS FALSE
C;    C IS TRUE
NOT(E);    E IS FALSE    D IS TRUE
$ END OF EXAMPLE $
```

Introducing new representations

This is an example to illustrate how a new type of object, in this case a rational number, can be introduced into ABSET and the definitions for its arithmetical manipulation given. These suffice to allow the system to do simple exercises in rational arithmetic.

We are not concerned here with efficiency – that would entail a discussion of compilation and choice of set representations. We are concerned with the points about clarity and style and the suitability of ABSET for descriptive programming and interactive use that were made earlier.

A further point made in the example is the use of MARRY to define functions over a union of domains in such a way that this union can be extended to include new domains such as fractions after the definition of the function and even after its application.

Finally the example illustrates how ABSET can be integrated into a practical disc-based operating system, and ABSET structures saved on disc.

The ability to save structures in modular fashion is based on the availability of an important data structure used in evaluation, namely, the interpreter's 'environment' or list of names and references to corresponding objects. Thus when saving any structure (using NEWDISC) it is not necessary to copy any of its substructures which are named in the environment as they can be saved separately. Further, the environment can be used by the interpreter to tell whether a copy of an object saved on disc is present in store (unwanted copies are garbage-collected) and to form one if it is not. This copy is formed using the latest saved versions of the named substructures of the object, which makes it easy to provide improved versions of system or other routines without having to recompile all the user programs referring to them, and is extremely handy for interactive work where one is continually making modifications.

Further the environment may itself be considered to consist of a number of named dictionaries, each belonging to a particular project and these may themselves be saved on disc (by typing SAVEPROJ in that project). Such information is safe against storage corruption or machine switch-off and is used to recreate the environment when the user next LOGS-IN to the project. Other projects having that dictionary in their environment will get the latest version when they LOG-IN next and thus two or more users may share information through a common library project. However information saved in separate projects is saved in separate dictionaries and this is very useful for interactive working as projects can be developed separately without interference.

```
$ THIS IS AN EXAMPLE OF AN INTERACTIVE SESSION WITH ABSET, IN
WHICH FRACTIONS(RATIONALS) ARE INTRODUCED AND MANIPULATED.
FOLLOWING SYSTEM RECOGNITION, THE USER LOGS-IN TO PROTOTYPE
PROJECT BASP, COPIES IT AND LOGS-IN TO THE COPY $
ABSET LISTENING TO YOU
LOG-IN BASP    COPROJ FRACT-ARITH;
LOG-IN FRACT-ARITH;
2*A'=6;    A...    3
$ INTEGER MULTIPLY AND DIVIDE PRESENT IN BASIC PROJECT $
A ISIN INTEGERS...    TRUE
2*A'=7;
A ISIN INTEGERS...    FALSE      $ 2 INTO 7 WONT GO $
$ FORMING HIGHEST COMMON FACTORS BY SET MANIPULATION $
ALLFACS'=MAP I' : INTEGERS TO
  (THOSE FAC' : 1 UPTO I SATISFYING QUOT' ISIN INTEGERS
  WHERE FAC*QUOT=I) : SETS-OF(INTEGERS);
ALL24'=ALLFACS(24);
PRINTSET(ALL24);
  [24, 12, 8, 6, 4, 3, 2, 1]
NEWDISC ALLFACS;
$ THE USER MAKES A COPY OF ALLFACS ON DISC AND PUTS IT INTO THE
DICTIONARY OF PROJECT FRACT-ARITH. $
PRINTSET(ALLFACS(6));      $ ALLFACS IS UNAFFECTED BY COPYING $
  [6, 3, 2, 1]
SAVEPROJ;
```

```
$ THE USER SAVES A COPY OF THE PROJECT DICTIONARY ON DISC, TO BE
USED WHEN LOGGING-IN. THIS SUPERCEDES FORMER COPIES $
NEW GREATEST', HIGHEST';
HIGHEST=MAP S' : SETS-OF(INTEGERS) TO SIT(S, O, GREATEST) : INTEGERS;
NEWDISC HIGHEST;
$ NOTE HIGHEST CAN BE COPIED TO DISC EVEN IF GREATEST IS
UNDEFINED $
GREATEST=MAP N', H' TO IF POSINT(H—N) THEN N ELSE H;
$ POSINT=+VE INTEGER $
GREATEST ISIN INTEGERS><INTEGERS→INTEGERS;
$ THE SPACE DEFINITION USES><AS INFIX FOR CARTESIAN PRODUCT,
AND A→B FOR THE SET OF FUNCTIONS FROM A ONTO B $
NEWDISC GREATEST; SAVEPROJ;
HCF'=MAP X', Y' TO HIGHEST(ALLFACS(X) CAP ALLFACS(Y));
HCF ISIN INTEGERS><INTEGERS→INTEGERS;
NEWDISC HCF;
HCF(15, 12)...    1
$ WRONG VALUE OF HCF(HIGHEST COMMON FACTOR) DUE TO ERROR IN
GREATEST $
GREATEST'=MAP N', H' TO IF POSINT(N—H) THEN N ELSE H;
$ CORRECTED $
GREATEST ISIN INTEGERS><INTEGERS→INTEGERS;
NEWDISC GREATEST;
SAVEPROJ;
$ THE USER LOGS-IN TO A FRESH VERSION OF HIS PROJECT CONTAINING
FRESH COPIES MADE FROM DISC OF ALL SAVED OBJECTS, INCLUDING
HIGHEST, HCF AND THE CORRECTED GREATEST. THEY ARE BOUND
TOGETHER SO THAT HCF USES THE LATTER AND HAS BEEN CORRECTED
WITHOUT RECOMPILATION.$
LOG-IN FRACT-ARITH;
HCF(15, 12)...    3
PRINTSET(ALL24);
SYNTAX ERROR      $ ALL24 WAS NOT SAVED ON DISC $
$ INTRODUCING </>, THE INFIX CONSTRUCTOR FOR FRACTIONS $
</>'=MAKEOP(401, 400, MAKEFRAC' : INTEGERS><INTEGERS→FRACTIONS');
NEWDISC </>;
$ DEFINITION OF INTEGER DIVISION TO GIVE THE CANONICAL FORM OF
THE CORRESPONDING FRACTION. $
CANFRACS'=THOSE N'</>D' : FRACTIONS SATISFYING HCF(N, D)=1;
DIVIDE'=MAP N' : INTEGERS, D' : INTEGERS TO NN'</>DD' : CANFRACS
  WHERE N=NN*H' AND D=DD*H AND H=HCF(N, D);
/'=MAKEOP(401, 400, DIVIDE);
NEWDISC DIVIDE; NEWDISC CANFRACS; NEWDISC /;
$ A FUNCTION TYFUN TO TYPE FRACTIONS $
TYFUN'=MAP X'</>Y' TO TYPE(INTCH(X)<>"</>"<>INTCH(Y));
$ INTCH(1)="1" $
TYFUN ISIN FRACTIONS→BOOLEANS;
NEWDISC TYFUN;
$ MAKEFRAC USES LISTS TO REPRESENT FRACTIONS $
MAKEFRAC=MAP N', D' TO['FRACT', N, D];
FRACTIONS=THOSE P'&Q' : LISTS SATISFYING P='FRACT'
  AND Q ISIN LISTPAIRS(INTEGERS, INTEGERS);
NEWDISC MAKEFRAC; NEWDISC FRACTIONS; SAVEPROJ;
$ A NEW EMPTY PROJECT IS MADE HAVING FRACT-ARITH AS ITS LIBRARY $
BUDPROJ FRACT-EXPT;
LOG-IN FRACT-EXPT
TYFUN(6/8);    3</>4
DELETE TYFUN;
TYFUN(12/18);    2</>3
```

```
$ TYFUN DOESN'T BELONG TO THIS PROJECT AND SO CAN'T BE DELETED $
$ DEFINITION OF ADDITION (AND SUBTRACTION) FOR FRACTIONS, USING
A COMPOSITION OF 3 FUNCTIONS TO ALLOW SIMPLE INFERENCE $
LOG-IN FRACT-ARITH
TYPE(TAPE('ADDF-20-07-70'));
$ THE TAPE HEADED 'ADDF-27-07-70' HAS BEEN READ BY THE SYSTEM AND
STORED $
ADDF'=MAP AB' WHERE AB=A'</>B', XY' WHERE XY=X'</>Y' TO PQ'
  WHERE PQ=P'</>Q'
    AND PQ=(A*Y+B*X)/(B*Y)
    AND XY=(P*B-A*Q)/(B*Q)
    AND AB=(P*Y-Q*X)/(Q*Y);
++'=MAKEOP(581, 580, ADDF: CANFRACS><CANFRACS→CANFRACS);
LOAD 'ADDF-20-07-70';      $ THE TAPE IS LOADED BY THE INTERPRETER $
NEWDISC ADDF;
NEWDISC++; SAVEPROJ;
LOG-IN BASP
1/2++3/4=A';
$ ADDING ++ TO FRACT-ARITH DOES NOT ADD IT TO BASP, AS BASP
WAS A COPY $
SYNTAX ERROR
LOG-IN FRACT-EXPT
1/2++3/4=5/Z'; Z...    4
$ ++ IS NOW AVAILABLE TO FRACT-EXPT EVEN THOUGH IT HAS BEEN
ADDED TO THE LIBRARY SINCE THE PROJECT WAS MADE $
$ AN EXAMPLE OF SIMPLE EQUATION SOLUTION IN FRACTIONS $
4/X'++6/8=5/4;
1/2++Y'=10/X;
TYFUN(Y);
3</>4
NEWDISC Y; SAVEPROJ;
COPROJ FRACT-TYPE;
$ FRACT-TYPE SHARES FRACT-ARITH AS LIBRARY WITH FRACT-EXPT,
BUT IS OTHERWISE INDEPENDENT $
LOG-IN FRACT-TYPE
TYFUN(Y);
3</>4      $ IT CONTAINS A COPY OF Y $
$ THE PRESENT TYFUN CANNOT BE EXTENDED TO TYPE INTEGERS ETC...,
SO A NEW ONE IS DEFINED (IN FRACT-TYPE) WHICH CAN $
TYFUN'=TYPE.STRFUN';
$ COMBINATION OF FUNCTIONS: THUS TYFUN(X) IS TYPE(STRFUN(X)) $
STRFUN=MARRY(STRFUNL') AS FUNCTIONS;
$ MARRY FORMS A UNION OF FUNCTIONS, THUS MARRY(STRFUNL, X)
WHERE STRFUNL=[SPACE1', FUN1']&[SPACE2', FUN2']&ETC, SELECTS FROM
THE ABOVE LIST THE FIRST FUNCTION 'FUN' APPLICABLE TO A 'SPACE'
IN WHICH X LIES AND APPLIES IT. THE DEFINITION OF STRFUNL MAY BE
GIVEN PIECEMEAL AS CONVENIENT, AND THE EVALUATION OF TYFUN
PROCEEDS AS FAR AS THIS ALLOWS $
STRFUN=[INTEGERS, POSTR']&STRFUNC';
TYFUN(1);      $ RESULT AWAITS DEFN. OF POSTR $
POSTR=MAP I' TO IF POSINT(I) THEN "PLUS"<>INTCH(I),
  ELSE "MINUS"<>INTCH(0-I);
POSTR ISIN INTEGERS→STRINGS;
PLUS 1      $ ABSET EVALUATION OF TYFUN(1) ABOVE COMPLETED $
TYFUN(Y);      $ STRFUN AS YET UNDEFINED FOR FRACTIONS $
TYFUN(-5);    MINUS 5
STRFUNC=[FRACTIONS, FRASTR']&STRFUNC';
FRASTR=MAP N'</>D' : FRACTIONS TO
  (STRFUN(N)<>"/"<>STRFUN(D)) : STRINGS;
```

```
PLUS 3/PLUS 4        $ ABSET EVALUATION OF TYFUN(Y) $
TYFUN(3-7);    MINUS 4
TYFUN(2/3++1/4);    PLUS 11/PLUS 12
$ END OF EXAMPLE $
```

Text in ABSET: illustrated examples

The following ABSET program illustrates the ideas discussed in the section 'Text in ABSET' in the earlier part of the paper.

It is worth emphasizing: (1) the point made at the end of the simple illustration of a directed graph, that is, the ease with which complex structures with circularities can be specified with the descriptive power of ABSET; (2) the elegance of the ABSET set construction in printing the constraints on an object obtained after partial evaluation of program relevant to the object.

```
$ A DEMONSTRATION OF SOME ABSET TEXT FACILITIES $
$ ... GENERATES TEXT USING THE CURRENT ENVIRONMENT TO NAME
SUBSTRUCTURES $
A'=[1, 2]; B'=[3, 4];
LL'=[A, [B, 10]];
LL...
[A, [B, 10]]
DELETE B;
$ DELETES THE IDENTIFIER 'B' FROM THE CURRENT ENVIRONMENT $
LL...
[A, [[3, 4], 10]]
$ THE SUBSTRUCTURE [3, 4] IS NO LONGER NAMED IN THE CURRENT
ENVIRONMENT AND ... GENERATES IT IN FULL DETAIL $
L'=[[1, 2, 3, 'LIST', [4, 5, 6], "END OF LIST"], [[1, 2], [[], 12]]];
L...
[[1, 2, 3, 'LIST', [4, 5, 6], "END OF LIST"], [[1, 2], [[], 12]]]
$ => GENERATES TEXT AT CONTROLLABLE LEVELS OF DETAIL $
L=>
BEGIN
[L2', L5']
$ NAMES HAVE BEEN INVENTED FOR THE SUBSTRUCTURES, AND HAVE
BEEN USED IN THE PRINTOUT: A NEW SCOPE LEVEL HAS BEEN INTRODUCED
FOR IDENTIFIERS AS INDICATED BY THE "BEGIN": ... CAN NOW BE USED
TO EXPLORE PARTS OF THE STRUCTURE $
L2...
[1, 2, 3, 'LIST', L1', "END OF LIST"]
L1...
[4, 5, 6]
$ THE "END" FACILITY ENABLES US TO REMOVE THE INVENTED NAMES
BY RESTORING THE SCOPE TO ITS ORIGINAL LEVEL $
$ ANY OF THE NEW NAMES CAN BE PROMOTED AND RETAINED BY "END" $
END L2;
L2...
[1, 2, 3, 'LIST', [4, 5, 6], "END OF LIST"]
$ THE SUBSTRUCTURE [4, 5, 6] IS NO LONGER NAMED $
L5...
SYNTAX  ERROR
$ AN ATTEMPT HAS BEEN MADE TO USE A NAME 'L5' WHICH IS NO
LONGER AVAILABLE $
$ STRUCTURES CONTAINING NAMED CIRCULARITIES CAN BE SAFELY
PRINTED OUT WITH... $
```

```
L'=[L, L, L, L];
L...
[L, L, L, L]
$ IN THE CASE OF UNNAMED CIRCULARITIES, => HAS TO BE USED $
BEGIN
L'=1&LL';
LL=[LL, LL, LL];
END L;       $ PROMOTE 'L' AND DELETE 'LL' $
$ L CAN SAFELY BE PRINTED OUT USING => $
L=>
BEGIN
1&L1'
L1...
[L1, L1, L1]
END;
$ A FURTHER EXAMPLE ILLUSTRATING THE ABILITY TO PRINT OUT TEXT
REPRESENTING STRUCTURES WITH CIRCULARITIES $
$ WE WANT A MACHINE REPRESENTATION OF A DIRECTED GRAPH: THE
GRAPH IS TO BE REPRESENTED AS A LIST OF NODES, EACH NODE BEING
A LIST OF WHICH THE HEAD IS AN INTEGER LABEL FOR THE NODE, AND
THE TAIL IS THE LIST OF NODES TO WHICH IT IS CONNECTED BY AN
OUTGOING ARC $
BEGIN
GRAPH'=[NODE1', NODE2', NODE3', NODE4', NODE5'];
NODE1=[1, [NODE2, NODE5]];
NODE2=[2, [NODE3, NODE4]];
NODE3=[3, [NODE1, NODE5]];
NODE4=[4, [NODE2]];
NODE5=[5, [NODE3]];
END GRAPH;       $ REMOVE THE NAMES NODE1, ....., NODE5 $
$ NOTE THE EASE WITH WHICH SUCH CIRCULARITIES CAN BE
CONSTRUCTED IN ABSET $
$ ALTHOUGH GRAPH CONTAINS CIRCULARITIES, IT CAN SAFELY BE
PRINTED OUT USING => $
GRAPH=>
BEGIN
[L6', L4', L8', L2', L10']
$ ANY FURTHER DETAIL REQUIRED CAN NOW BE OBTAINED USING... $
L6...
[1, L5']
L5...
[L4, L10]
$ ETC. $
END L6, L4, L8, L2, L10;
$ THE NEW NAMES FOR THE NODES CAN BE PROMOTED $
GRAPH...
[L6, L4, L8, L2, L10]
L6...
[1, [L4, L10]]
$ =>> GENERATES TEXT WHICH GIVES ALL THE CONSTRAINTS ON A
NAMED OBJECT IN THE FORM OF AN ABSET SET CONSTRUCTION $
$ A POSSIBLE METHOD OF INTRODUCING FRACTIONAL ARITHMETIC TO
ABSET WOULD BE TO DEFINE "NUM" (NUMERATOR), "DENOM"
(DENOMINATOR), AND "MAKEFRAC" (MAKE FRACTION) RELATIONS AS
FOLLOWS $
NUM'=MAP A'&B' : INTEGERS><INTEGERS TO A : INTEGERS;
DENOM'=MAP A'&B' : INTEGERS><INTEGERS TO B : INTEGERS;
MAKEFRAC'=MAP NUM(X') : INTEGERS, DENOM(X) : INTEGERS TO
  X : INTEGERS><INTEGERS;
```

```
Z'=MAKEFRAC(1, 2);
Z...    1&2
$ WE NOW REQUIRE AN EQUIVALENCE RELATION FOR USE IN
ASSERTIONS ABOUT EQUALITY OF FRACTIONS $
FRACTEQ'=MAP X' : INTEGERS><INTEGERS, Y' : INTEGERS><INTEGERS TO
(NUM(X)*DENOM(Y)=NUM(Y)*DENOM(X)) : BOOLEANS;
FRACTEQ(MAKEFRAC(2, 3), MAKEFRAC(4, 6))...
TRUE
$ =>> CAN BE USED TO PRINT OUT THE CONSTRAINTS ON AN OBJECT
AS SHOWN $
FRACTEQ(X, MAKEFRAC(2, 3));
X=>>
THOSE VAR2'&VAR1' : INTEGERS><INTEGERS SATISFYING
MULT(VAR1,2)=VAR3' AND MULT(VAR2, 3)=VAR3
$ THIS PRINTOUT SAYS THAT X IS IN THE EQUIVALENCE SET OF
FRACTIONS WHICH ARE EQUAL TO 2/3 $
$ END OF EXAMPLE $
```

CONCLUSIONS

The work described here concerns the declarative part of ABSET. We consider that this is deficient in two main respects and it is in these areas that we are attempting improvements.

First, although we can write new sequencers we shall need further new primitives and new syntactic constructions to make this convenient. In particular we need to treat assignment, reference, and storage allocation.

Secondly, we need to be able to recognize when more sophisticated sequencers are needed. Using a complex sequencer (for example, keeping records for back-tracking) where a simple one would suffice can be very inefficient. However, we need to be able to call an appropriate complex sequencer into play. This appears a difficult recognition process at present.

Acknowledgement

The work reported in this paper is sponsored by the Science Research Council.

REFERENCES

Elcock, E.W. (1968) Descriptions. *Machine Intelligence 3*, pp. 173–9 (ed. Michie, D.). Edinburgh: Edinburgh University Press.

Fikes, R.E. (1970) REF-ARF: A system for solving problems stated as procedures. *Art. Int.*, **1**, 27–120.

Foster, J.M. (1968) Assertions: programs written without specifying unnecessary order. *Machine Intelligence 3*, pp. 387–91 (ed. Michie, D.). Edinburgh: Edinburgh University Press.

Foster, J.M. & Elcock, E.W. (1969) ABSYS 1: an incremental compiler for assertions: an introduction. *Machine Intelligence 4*, pp. 423–9 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Murray, A.M. & Elcock, E.W. (1968) Automatic description and recognition of board patterns in Go-Moku. *Machine Intelligence 2*, pp. 75–88 (eds Dale, E. & Michie, D.). Edinburgh: Edinburgh University Press.

Nevins, A.J. (1970) A programming language with automatic goal generation and selection. *J. Ass. comput. Mach.*, **17**, 216–30.

Robinson, J.A. (1969) Mechanizing higher-order logic, *Machine Intelligence 4*, pp. 151–70 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L. & Koster, C.H.A. (1969) *Draft report on the algorithmic language ALGOL 68*, *MR100*, Amsterdam: Mathematisch Centrum.