Section Three

Design

Contents

1 Design Features

1.1 Design Aims

Before attempting to design the programming language, the team discussed what features each member felt were important and drew up the following short list of design aims:

- The language should use simple semantic and syntactic models.
- The syntax should follow a natural English language format.
- The language should be statically type checked as far as possible.
- The object oriented model should be simple, yet powerful enough to enable the full range of object oriented modelling techniques to be used.
- The language should support a persistent store, to provide a type secure method of storing program data and sharing such data between programs.

From the list of design aims an abstract prototyping language was created. The prototype language gave a reference point on which the various design aims could be developed and modified. This abstract language also enabled the team members to convey their ideas more clearly to each other. Once the final design aim list had been verified, the abstract language was discarded and the process of designing the Anubis programming language began in earnest.

1.2 Type Systems

The overriding desire in the design of the language was to make it as statically type checked as possible. It was felt that type errors should be given to the programmer during the design/compilation stage of a program than to the program user at run-time. It is possible for dynamic errors to go undetected until a program is in serious use. When such an error arises, it can be not only confusing for the program user but also a danger to the integrity of the data produced/used by the program. Therefore, the design of the language revolved around a semantic model that allowed such features to be implemented easily. Previous object-oriented language designs relied heavily on dynamic type checking of objects, especially regarding the inheritance model. With the strong bias on static type checking, Anubis evolved an inheritance model that, as far as possible, allowed the objects to be statically type checked.

2 Object Models

In designing the language, the team had to discover what features truly described object-oriented programming. Of the many conflicting models reviewed a pattern of common features evolved:

- Encapsulation – the creation of a block, containing all the definitions of the operations allowed on a block (object) of that type.
- Inheritance – the mechanism by which objects can be related to each other with shared operations.
- Polymorphism – the ability for the same operation to be defined over different object types.

Research into existing object-oriented languages resulted in the decision to include multiple inheritance and inclusion polymorphism in the design of Anubis. In designing the multiple inheritance model, the team developed a unique and simple method for handling the name clash problem associated with multiple inheritance schemes. The actual method will be described in detail later in the design section.

The research uncovered two fundamentally different models of implementing object-orientation in a language: the completely object- oriented approach and the multiple base type and constructor model. Each model although object-oriented in nature, has an entirely different philosophy. Since the specific model used by the language is crucial to both its semantic and syntactic appearance, each model had to be studied in depth and a decision of its adoption made only after all the relevant factors had been considered. In keeping with the importance of the decision, the models will be discussed separately below.

2.1 Completely Object-Oriented Model

This model proposes that every item in the language is an object, resulting in a consistent method for handling all data types. Data items such as integers, reals, strings etc. are all objects and have the same method of applying their operators. The model often involves a message passing syntax for operator (method) application, in which a message to apply an operator is sent to the required object. This model is used by Smalltalk which is often attributed as being the only true object-oriented programming language since it does not differentiate between base types and objects, and all types are objects.

2.2 Multiple Base Types and Constructors

This model consists of base types (integers, reals, strings etc.) that form the language foundations and objects, which are new types, constructed from the base types. The model allows object-oriented techniques to built on existing language designs since it requires the base types to have been previously defined. An example of such a language is C++, which extends the base types of C. Often the base types are re-defined and new ones added to aid an elegant implementation of the object-oriented model. For example, C++ defines a new type stream for input and output that uses message-passing semantics, enhancing the object-oriented approach of the language. The advantage of this model is that it is easy for a programmer conversant in an existing language to extend their techniques to include object-orientation, than learn a completely new method of designing and implementing software.

2.3 Selection of Object Model

Originally the completely object-oriented model appeared the most appealing owing to the consistent method of handling all data types. After more detailed examination, however, the model became conceptually complicated to implement, with grave concerns regarding the grounding of recursion. These worries also extended to the problems that a programmer may have in understanding the use of the model. Therefore, it was decided that the base type and constructor model was more in keeping with the original design aims and was the object-oriented model adopted by Anubis. This model leant itself well to being implemented as an imperative language, and so this was chosen over a functional model.

3 Persistence Model

The original design specification included the need for features to support data base programming. The experienced gained by the team members working on programming database systems, coupled with the experience of the PS-algol Persistent Database Model, suggested that support for persistence by the language would help in the creation of such systems. The design requirements of the persistent store model were:

- The semantics of the model should be easily understood by the programmer.
- The operations allowed over the store should not restrict the programmer in the modelling of the database structure.
- The model should be easily accommodated within the semantics of the language.

In satisfying the first requirement, the model was designed as a logical extension of the non-persistent run-time heap. The model proposed that the persistent store was no more than a heap, shared by more than one program, that persists beyond the run-time of the programs. At the time of conception, it appeared that a flat name space model would be the only model that could be implemented in the time scale of the project. During the development and implementation of the store, it became clear that the store design would support a layered persistent model akin to the Napier88 environment model. This however, would involve changes to the syntax of the language and hence the compiler. The time scale of the project means that the changes cannot be implemented at present.

The second requirement was met since the persistent store exactly models the run-time heap and there are no semantic differences between the handling of persistent and non-persistent data structures. All operations allowed on non-persistent types are also allowed on persistent items, therefore, the persistent store model does not impose any restrictions upon the programmer that are not already present in the language.

The final requirement was met by ensuring that the persistent store was regarded as an extension of the run-time heap. The modelling of the store in this manner meant that it was logical for the interface procedures to operate in a method consistent with the access of items in the non-persistent run-time heap.

4 Design Process

4.1 Semantics vs. Syntax

Any programming language can be viewed at several levels, including a semantic level and a syntactic level. The syntax of a language is the actual text used to write a program in that language, whereas the semantics is the meaning associated with the text. Since syntax is the medium of communication of semantics, it is impossible to discuss semantics without using some form of syntax. Although there exist systems for representing semantics, they are often cumbersome and ill suited for the representation of a particular language. Throughout this section language constructs will be covered through descriptions, with occasional reference to example syntax. This is not intended to reflect the actual syntax of Anubis, but merely indicates the sort of use to which a construct would be put.

4.2 Design Principles

Three design principles were used during the design of Anubis, in order to make the language as complete as possible. These principles have been refined over a number of years, and are intended to minimise the inconsistencies and idiosyncrasies of a language. The use of these principles as a basis of a design methodology does not mean that they cannot be broken. However, each exception must be well justified, as the principles help keep a language simple, and thus powerful.

4.2.1 The Principle of Correspondence

The principle of correspondence states that all the rules governing the use of names in a language should be designed together so that names are always used consistently in programs. An example of this is that the scope rules should be the same everywhere. In most algol-like languages names may only be introduced within declarations and as procedure parameters. The principle of correspondence says that there should thus be an equivalent parametric declaration for every type of declaration in the language.

4.2.2 The Principle of Abstraction

Abstraction is a method of identifying the essential structure of something to hide the unimportant details. The principle of abstraction states that it is useful for the programmer to be able to abstract over language constructs, such that the underlying structure can be defined, and then used repeatedly. An example of this is the use of functions to abstract over expressions.

4.2.3 The Principle of Data Type Completeness

The principle of data type completeness states that all data types should have the same 'civil rights' in the language, and that the rules for using data types should be complete, with no exceptions. Thus any general operation, such as assignment or parameter passing, should apply to every data type.

4.3 Design Theory

The design approach used was based upon that used in the development of S-algol. This technique is based upon five basic stages, which are iterated to refine the language gradually.

4.3.1 Data Types

The first stage is to define what data types are allowed in the language, and what is meant by them. The operations on these types should also be defined. This is an important step, as the types available directly affect the uses to which the language can be put. The principle of data type completeness should be noted, as any inconsistencies in the use of the types makes the language more complex and usually less powerful.

4.3.2 The Store

If a store is to be used in the language, its semantics should be declared. The relationship between data types and the store should be investigated, including the semantics of pointers, locations and constants.

4.3.3 Abstraction

The abstractions of the language should now be investigated. The semantically useful constructs in the language should be identified, and abstractions should be identified for each. This is not always easy to achieve.

4.3.4 Declarations and Parameters

Due to the principle of correspondence, declarations and parameters should be considered together. The syntax does not have to be the same, but there should be equivalent semantics. This equivalence includes whether the language has call-by-value or call-by-reference semantics.

4.3.5 Input and Output

Finally, the input/output model should be introduced. This is the interface with the outside world, and thus is more restricted in its design than the other areas. Care must be taken to ensure that the model is reasonably

portable across different environments, or has sufficient flexibility to be changed according to the machine being used.

4.3.6 Iterate

These five steps should now be repeated, to correct any problems or inconsistencies in the language's design. This should continue until satisfactory results are achieved.

4.3.7 Concrete Syntax

Once the full semantics of the language are finalised, a concrete syntax for the language can be designed. This should take into account ease of use and readability, and have no ambiguities. A good balance can be achieved for general-purpose use, or a variety of different forms can be developed for different applications.

4.4 The Design Process

4.4.1 Base Types

In accordance with the design principles outlined above, the first stage was to decide upon the types Anubis should have. It had already been decided to adopt a model using a set of pre-defined base types and constructors, with the ability for the user to introduce his or her own new types. The full semantics of the user defined types had yet to be determined, but it was known that these 'object' types would allow grouping of information into some kind of structure, with operations defined over this information. This had to be borne in mind when introducing the base and compound types, as it is inelegant to have built-in types, which could easily be defined within the language. (Having said this, it arguably allowable to have such duplications, as the pre-defined version can be optimised for efficiency.)

The first types introduced were integers and reals. There was a certain amount of feeling that there should be a single generic type 'number', but this concept was conceded on the same grounds as for every other language: efficiency of implementation. It was decided not to allow integers and reals to be used interchangeably (for instance, passing an integer into a function expecting a real), but the division operator acting on two integers would return a real result. The user could thus explicitly pass an integer value into a real expression by this means.

Next the boolean type was introduced. While many languages cope without this type, it is difficult to justify this standpoint. The lack of booleans forces conditional operations to be 'sealed units', without being able to sub-divide the operation into typed parts. Hence equality would not be a typed

concept, just a subpart of the conditional operator. It was felt that the idea of having booleans as a type in their own right was much more elegant, and would give the language the power to introduce abstractions over conditional expressions.

Characters and strings were the next types to be considered. Textual manipulation is so important in today's programs that good dynamic string handling was felt to be particularly necessary. It was decided to use strings only, though, because it was felt too cumbersome to treat strings as a constructor acting upon a character type. Character manipulation is required, but characters may be represented as a string of length one. Strings of greater length are used so much more frequently than characters to justify the choice of strings as the only textual data type.

For the same reasons, it was decided to limit graphical representation to images (being two dimensional arrays of pixels). Again, it would have been possible to regard images as being constructed from pixels, but the pixel on its own is of little use, and can be represented by an image with unary dimensions anyway. It was decided not to complicate the language with multiple plane graphics for the first release. Much discussion was also entered into as to whether or not a special representation for images was needed. An idea proposed was to regard images as a two dimensional array of booleans, which could be the internal representation of an object. However, unless this type built was into the language and optimised in the compiler, this method would be very inefficient, which is not desirable for graphics operations. Since it would be built into the language anyway, it was felt to be more consistent if it were implemented as a base type.

Other graphics types were investigated, such as the Outline graphics system in S and PS-algol. These were rejected, in that they are prime candidates for user defined object types. An object type for line graphics could allow the user considerably more flexibility, including the updating of the contents of a line drawing, and three-dimensional representation, with interfaces to map the drawing to a two-dimensional image.

The final base type considered was that of files, or streams. This would allow a reasonably generic method of accessing the outside world, although the operations on the type would need to have some dependency on the hardware used. This type was rejected on the grounds that it was largely unnecessary, and could undermine the integrity of the type system. From the point of view of storing information, the persistent store would be far more useful (holding *all* data types), and would be type safe (a value is guaranteed to be of the same type when it is read back in). For the purpose of interfacing with data from other sources, it could be useful to have access to files. However, the model for input/output, however, would allow this, and if it was really felt necessary to have access to files, this could be implemented as standard procedures.

4.4.2 Type Constructors

Having defined the base type, the type constructors in the language were considered. These would extend the language to give an infinite type system. Any decent language should have at least one method of grouping information together, and the simplest way of achieving this is through an array or list. Both were investigate to determine whether or not they would be applicable or needed.

The list is one of the most intuitive forms of data structure for humans, possibly because its linear nature corresponds well with the linear nature of time. People tend to make lists of items, adding one item at a time, as they think of them. This also emphasises the dynamic nature of lists: it is always possible to add another item to a list. The constructor of the list rarely knows how many items are in the list: if he or she needs to know, then the number of items is counted at the end, or a count is kept as each item is added. This dynamic nature conflicts somewhat with the design of computers. Although computers are intended to represent dynamic information, memory management dictates that it is easier to change the value of the information than the size of it. Compromises have to be made in the storage of a list, with the result that it is difficult to ascertain where a given element of a list is stored.

It is this conflict that has allowed the concept of the array to dominate the world of imperative programming languages. This compromises the flexibility of the data structure to take account of the efficiency inherent in having a model very close to the actual hardware. As a result the array is efficient, allowing direct access to each of its elements, and thus it also has a strong concept of ordering. The sacrifice is at the expense of dynamicity, for arrays are generally fixed in size, and therefore static.

It was these compromises that lead to the wish to include both lists and vectors in Anubis (a vector is a single-dimension array). Every programmer has written functions to act over lists, and these occur in a significant number of programs. Much work was thus done on finding an elegant model for a list within an imperative language. The most promising model was that commonly used in functional languages, whereby a list can be considered as having a ‘head’ (the first item in the list) and a ‘tail’ (the rest of the list). The model included the ability to create an empty list of a given type (all elements in a list have to have the same type, so that list de-referencing could be correctly type checked), or construct a list using a list constructor operator acting over a syntactic list of data items. The operators supported were to append two lists, to extract the head of a list, and to obtain the tail of a list. Thus valid operations (using an example syntax) might be as follows:

```
let a_list := nillist of int
a_list := a_list ++ [ 1, 2, 3, 4, 5 ]
write head( a_list )
write head( tail ( a_list ) )
```

A final idea was to allow indexing into a list, much as is possible in languages such as SASL. As these various refinements were made, it could be seen that the two models for vectors and lists were gradually converging. With this final refinement, lists suddenly offered everything the vectors could provide semantically. The only difference was that the extra features supported in lists meant that a different method of implementation was necessary, which would be less efficient than vectors on the overlapping features. This left a dilemma, in that should this duplication be allowed? The answer came from a different source, during investigation of other forms of redundancy. It was realised that the object model could handle the implementation of lists quite satisfactorily, with no loss of efficiency. Thus lists could be omitted from the basic language, but using the principles of object-orientation, a list handling mechanism could easily be written and placed in the persistent store for future use.

Two remaining type constructors were also considered: a form of set, and a record or structure type. Sets were discounted for exactly the same reasons as lists – they could easily be implemented on top of the language, and due to the object-orientation, used without inconvenience to the user. Structures were thought about, but it was soon realised that they were a poor cousin to the concepts of objects. Objects allow information to be grouped together in a similar manner to structures, but they also allow control to be exercised over how the information is used. Structures were thus unnecessary, and were omitted to encourage an object-oriented programming methodology.

It was realised that other constructs would be needed, such as object definitions and functions, which would also be type constructors. These were considered separately, however, as they are essentially methods of abstraction.

4.4.3 The Store

It was decided that Anubis should have a simple store model, with L-values (locations) being updated with R-values (the values stored in locations). With the scalar types, R-values would be their actual values, whereas with complex types, R-values would be the heap location where their full representation was stored (ie. pointers).

4.4.4 Abstractions

The first abstraction to be introduced was the idea of objects. This was one of the underlying concepts of the design of Anubis, being based around the

idea of defining new types in the language. Objects were envisaged as an abstraction over program sequences, having a number of declarations, forming an environment, which would vary with time. This could be accessed by some kind of interface, corresponding to the way a program sequence would be accessed by procedure calls. From this it followed that Anubis would need a functional abstraction of some kind (abstracting over expressions).

The functional abstraction chosen was that of procedures, which could either return a result, or act as a void statement. The body of the procedure would contain an expression calculated in terms of parameters passed into the procedure. A block mechanism could also be introduced, whereby a sequence of void statements followed by an expression, all acting upon a local environment, could be used in place of a typed expression. This would extend the usefulness of procedures, as they would be allowed to contain large sequences of calculations in order to calculate their result.

Now the concepts of functional abstraction had been decided, the question of interfacing to objects was looked at. The main point to be considered was whether or not is was necessary to have a new type for the interfaces to objects. Was there a semantic difference between functional abstraction ("procedures") and interfaces to objects ("methods")? On the one hand, procedures are scoped (in that they can be accessed whenever the identifier they are bound to is in scope), whereas methods are tied to a type, and can only be used with an object of that type. As a result a type 'method' would stop the language being data type complete, since methods would not be first class items in the language. This difference in scoping can be seen in this pseudo-code fragment:

```
object_type example is
{
        a = . . .

        procedure . . . . . a . . . .
        procedure . . . . . .

        method . . . . . a . . . . procedures
}
```

This shows a new type being introduced, called "example". It has an internal value called 'a', and two internal procedures, one of which accesses the value of 'a'. It has one method, which accesses the value of 'a', and also calls the two procedures. Thus methods have the same properties as procedures, within the type definition, but only the methods can be accessed outside the definition.

The distinction between procedures and methods was disliked, and so the possibility of eliminating one or the other was investigated. One idea that appealed was to have methods only, as this would enforce an object-oriented attitude to programming. However this would mean that all the functional abstractions within an object would be available for use outside the object. The protection of internal functions was thought to be a very important part of object-oriented programming. The alternative was to have procedures only. How would this allow objects to be interfaced though?

The solution was to remove the protection mechanism from the procedures themselves, and place it in the hands of the type definition. This would mean that a type definition would be a *protected block*, which could not be accessed from outside. There would be some syntactic mechanism for designating which of the procedures defined in the block formed the interface, and these would be known as *interface procedures*. As far as the scoping rules were concerned, the type definition would give just a list of procedures that were to be stored in any object of that type. Such an object could be de-referenced to obtain the interface procedure, which could then be called. The interface procedure, of course, would have access to all the internal values, in accordance with the normal scoping rules. Thus the simple concept of the protected block would enable the model to be implemented with just procedures, and without compromising the scoping mechanism at all.

4.4.5 Declarations and Parameters

The semantics of declaration state that an identifier must be declared before it is used. Thereafter that identifier is bound to a given location throughout the current environment. Within an inner environment, the identifier can be bound to a new location, the binding returning to its original state outside the local environment. The equivalent parametric declaration is to bind the values passed into a procedure to the identifiers declared in the local environment of the procedure (call-by-value). Note that some data types have pointer semantics, in that their 'value' is identity of a particular data structure.

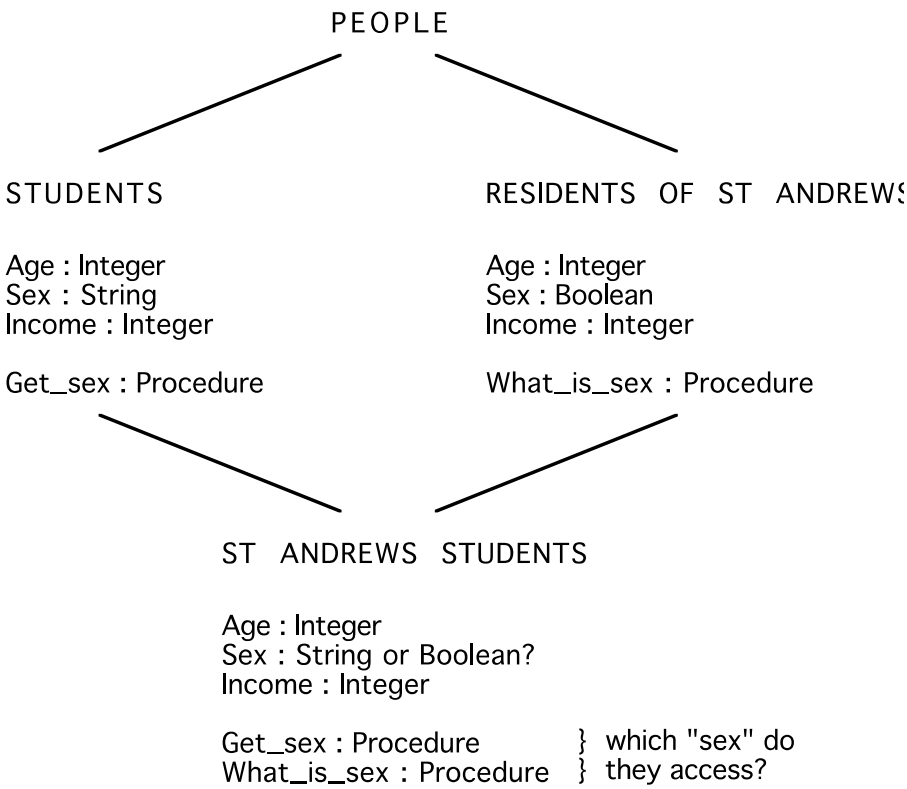
For the object abstraction mechanism, declarations in a standard environment have a direct counterpart in the object's environment. This extends to all data types, including procedures. The corresponding action to calling a procedure is to call an interface procedure. This procedure call has exactly the same semantics as a normal procedure call, and the procedure can return a result, or be void (simply having a side effect on the object in question).

4.4.6 Inheritance Model

The object model desired include inheritance, and in particular, multiple inheritance. Inheritance would allow the defining of object types based on

existing object types. Any internal representation from the old type would be included in the new type, and new features could be added. These features could, if required, re-define those from the old type.

Multiple inheritance goes one step further than this, and allows a type to inherit from more than one existing type. Thus two types, 'Students' and 'St Andrews Residents' could be used to define a new type: 'St Andrews Students'. Thus a type hierarchy would be created, of the form of a directed acyclic graph. Problems occur, though, if the inherited types both contain information with the same name. The various sets of internal representation cannot then be merged to create the internal representation for the new type. Consider, for example, the St Andrews Students case:



Here the definition of St Andrews Students is vague. Should the internal value 'sex' be a string or a boolean? Should there be one of each? Many solutions were considered:

(1) Automatically rename any internal values which have name clashes. Thus the two instances of 'sex' would become 'sex1' and 'sex2', or 'students_sex' and 'st_andrews_resident_sex'. The inherited procedures would still access the correct representation. This would be cumbersome, though, with name clashes with interface procedures. What if both types had a 'get_sex' procedure? Then any 'st_andrews_student' object would have two representations of its sex, and two corresponding ways of accessing it;

'st_andrews_residents_get_sex' and 'students_get_sex'. Also, what if the user initialised one value, but not the other? Result: one very confused student.

(2) Force the programmer to explicitly rename attributes when a name clash occurs. This is equivalent to approach (1), but ensures that the programmer thinks about the naming problem. This suffers from the same drawbacks, and also requires intervention by the programmer.

(3) Disallow name clashes altogether (so that multiple inheritance is only allowed if name clashes do not occur). This is not particularly satisfactory, particularly with the 'plug-in modules' approach that object-orientation allows.

(4) Associate internal values with specific interface procedures only. Thus each interface procedure would have unique control over its own set of values, which could not be accessed by other procedures. Thus this binding could still be kept even if the name was used again in a different inherited type. Interface procedure name clashes would still have to be handled in one of the above manners. This model severely limits the usefulness of the object model.

(5) The solution finally chosen was to impose precedence on the inherited types. Thus any name clashes would result in one of the items being chosen according to the priority specified by the programmer. Because of the strong type checking, this would only be allowed if the two items had equivalent types. The semantics would be that of overwriting existing definitions: each inherited type would be taken in turn, with any declarations overriding previous declarations with the same name. The declarations introduced explicitly in the type definition would override all others.

4.4.7 Iteration

The various components of Anubis were gradually refined over a period of time. The principle modifications were to the object model, as this formed the underlying structure of Anubis.

The most significant work done was with how the newly defined types would fit into the existing type system. To keep data type completeness, it was already decided that user-defined types could be used wherever pre-defined types could. The question was as to what relationship exists between similar types within the type hierarchy. Since a subtype just has more information than the type it inherits from, there is no reason why it cannot be used wherever the supertype was used. This simple inclusion polymorphism allows generic procedures to be written which act over a given type, which can then be used with any of that type's subtypes.

Similarly, if a procedure returns an item of a given type, then it can return an item of a subtype, since it will simply be returning more information than was required.

This scheme, regrettably, requires a certain amount of dynamic type checking. As much type checking as possible is done statically within the compiler, for it can check that if a value is passed into a procedure, then it is exactly the same type as is needed. If it is not, then a compiler warning can be given to inform the programmer that the procedure call will be dynamically type checked. An additional check can be made to ensure that the value can *at some point* be of a correct subtype of the value required. If this is never possible, then a static type error will result.

An extension of this idea was also experimented with. It was wondered whether or not identifiers could be allowed to bind to objects of different types at different points in the program. The problem was that to allow the types to be changed freely, even if restricted to subtypes and supertypes, would lose the ability to perform any static type checking on objects. Eventually an idea was proposed, whereby each identifier would have a stated type (strictly speaking, each *location* would be typed, as the same would apply to individual elements of a vector). Thereafter, the identifier could bind to an object of any subtype of the identifier's type. Thus the identifier can be type checked according to its designated type, since its actual value will always have the same or more information than its stated type.

An extension of the type hierarchy model was to introduce a root to the tree, called "any"; an object that did not specify any inherited types would automatically inherit this root type. Thus every object type would be a subtype of this root. The usefulness of this is that procedures could be written which could act over any object, without needing to know its type. Obviously any actions applied to such an object would require dynamic type checking. This approach is comparable to a pointer type (such as 'pntr' in S and PS-algol). Thus the object scheme can be made as vague as required, or tightened up by using strong typing, according to the application.

Another addition to the model was to allow an object access to itself. Formerly an object could read and update its internal representation, but not the whole object. The standard identifier 'self' was introduced within type definitions to allow internal procedures to access the enclosing object.

With an object model designed, a method now had to be developed for the creation of instances of an object. One method considered was to have a special 'constructor' for each type, specifying how to initialise a new object. This constructor would essentially be a special procedure that would return an object of the required type. The programmer would decide whether to use this procedure to give the object default values, or to accept parameters

to initialise the object directly. If a type did not have a constructor defined, then no instances of that type could be created, and it could only be used for inheritance.

This approach was rejected, however, in favour of simplicity and consistency. The model for declarations forces the programmer to give an initialising value for each type when it is declared. It was thus felt natural to use this mechanism to specify initial values for each internal value in a type definition. This ensures that each value is always defined before it is used. This method does not exclude the use of constructors as described above, but since an initialising operator could be defined by the programmer as an interface procedure, this was considered unnecessary. In instance of an object type could be created using the type name, being initialised with default values. These could then be updated to useful values by an application of an interface procedure.

4.4.8 Concrete Syntax

Having completed the semantic design process, a concrete syntax was developed. This was not a difficult process, as a good 'feel' for the language had been developed during the design process, and a syntax formed very naturally. The principle design aim for the syntax was that it ought to 'flow' well. Programmers try, often without success, to read their programs back to themselves. Often they are filled with phrases like "squiggle back-slash thingy ping!" which is not really very helpful. The desire for Anubis was to have a language that would consist mostly of English words, arranged in such an order as to allow natural reading. Even where symbols had to be used for conciseness, a suitable 'pronunciation' would allow the text to be quite readable. Some examples are as follows:

object person is [. . .]	The object "person" is . . .
object student inherits person is [. . .]	The object "student" inherits from person and is . . .
let fred := person	Let "fred" be a person
fred := student	"fred" becomes a student

The full syntax for Anubis is given in the next section.