Ada

Ada was developed in recognition of the need for modular and reliable programs. It introduced abstract data types supported by separable modules. Abstraction requires ***information hiding***, users have to access modules through an abstract interface (mathematical not implementation) which hid implementation details. The basic structure of the language closely followed Pascal.

Ada was first developed for DoD applications of *embedded computing*. To assure portability, the DoD did not allow the development of either subsets or supersets of the Ada language (this was later changed in Ada95).

Declarations

The most significant difference between Pascal and Ada is in ***declarations***, those non-executable statements in the front of a program which inform the compiler and other preprocessors about the semantics of the language. Ada declarations are of five types:

1. Object	constants and variables
2. Type	object types
3. Subprogram	functions and procedures
4. Package	(new) modules
5. Task	(new) modules which execute concurrently

Modules (packages and tasks) are disjoint environments which communicate through defined ***interfaces***. Module declarations have two parts: the interface specification and the body of the implementation. The central difference between a package and a block is that packages have names and formal parameters, while blocks do not.

Data Structures

Ada was the first to introduce ***floating-point*** and ***fixed-point*** number types. Floating-point numbers have round-off errors while fixed-point numbers have an absolute error bound.

Ada introduced new typing tools. ***Subtypes*** are subsets of a type. ***Constraints*** are restrictions on the members of a type which can be evaluated at runtime. ***Derived types*** foreshadowed object-oriented inheritance, they are types which inherit operations, functions, and attributes from a parent type.

Name Structures

The block structure of Algol still permitted global variables, in that blocks provided encapsulation of control but not of names. The problem was ***side effects***, which can be defined as hidden access to a variable. A related problem was ***indiscriminate access***, that is, no programming tools prevented access to variables, even when access was inappropriate. There was no way in block structured languages to prevent indiscriminant access. Yet another related problem was ***vulnerability***, there was no way to preserve access to a variable, in that a new declaration might intervene between an old declaration and the use of variable, blocking the scope of the old declaration. Finally, block structure permitted ***overlapping definitions***, that is, shared access to variables. This undermines modularity.

Parnas provided two principles of information hiding:

1. One must provide the user with all the information needed to use a module, and nothing more.

2. One must provide the implementor with all the information needed to complete the module and nothing more.

That is, the user cannot write programs which access the implementation details, while the implementor has no knowledge of the context of usage of the module.

The Ada construct which supports information hiding is the ***package***. This is achieved by having two separate components of a module, the interface and the implementation. Packages control name access by mutual consent: the package implementor nominates accessible variables by making them ***public***, while the package user ***imports*** a package when its public variables need to be used.

Packages are abstracted by the notion of a ***generic package***. Generic packages provide a template which can be instantiated by multiple instances of the package. However, generic packages are difficult to compile. Here is Ada code for a type independent generic module for stacks:

```
generic
  Length : Natural := 100;
  type Element is private;
package stack is
  procedure Push (X : in Element);
  procedure Pop (X : out Element);
function Empty return Boolean;
function Full return Boolean;
Stack_Error : exception;
end Stack;
```

`Element` is a ***type parameter*** which is declared to be private. Thus the package can be instantiated with stacks of different types. Here are two examples of construction of new stacks:

```
package Stack1 is new Stack (100, Integer)
package Stack2 is new Stack (256, Character)
```

`Stack1` can accommodate 100 integers, while `Stack2` can accommodate 256 characters.

Control Structures

Ada control structures are similar to those of Pascal. Since Ada was intended for embedded applications, it was important that Ada have ***exception handling*** capabilities. Ada permits definition of exceptional circumstances, and provides mechanisms for signaling their occurrence and responding to their occurrence. Although all other names in Ada are bound statically, exceptions are bound dynamically. (Thus exceptions are exceptional.)

Ada introduced ***position-independent parameters***, that is, parameters can be in any order. This is achieved by the simple expedient of labeling parameters with names. The names identify the parameter's function. As well, parameters could be given a ***default value***. These changes in the definition of parameters make compiling more complex.

Concurrency

Ada provides a ***tasking*** facility, which allows a program to do more than one thing at a time. Tasks that are both concurrent and in communication must be synchronized. Ada ***synchronization*** is very much like mutual procedure calls. When a task has some data to communicate to another concurrent task, it calls that task, passing the data as parameter bindings. The only difference is that the first task does not halt, rather it keeps on processing. Should a concurrent task need data before it is sent by another task, that task simply waits until the data is sent. Should a task send data before it can be received, the sending task again waits until the data is received before continuing. This type of coordination is called a ***rendezvous***; the communication regime is called ***synchronized communication***. Should a rendezvous fail to take place, both tasks may wait indefinitely; this is called ***dead-lock***.

Tasks are ***tightly-coupled*** when they mutually communicate, waiting in turn for data. Tight-coupling has the disadvantage that both tasks must process at nearly the same speed. That is, the speed of processing is limited to the slowest task. To ***loosely-couple*** tasks, a buffer must be inserted into the communication stream.

Malignant Growth

Ada grew into a language which was too large, about three times larger than either Pascal or Algol. This means that the language is difficult to learn and more difficult to manage. Increase in language size can be viewed as a kind of entropy, causing the design to deteriorate over time. Another term for this is ***featuritis***, a phenomenon which is prevalent in committee designed languages. The benefits of adding a feature appear to outweigh the cost of adding a small increment to the language. Benefits seem clearer and easier to justify since they are small changes. However, their accumulated effect is a global negative. Features are by definition added piecemeal, independent of consideration of the entire language. This leads not only to excessive size, but also to feature interaction which can, at worst, increase complexity and errors exponentially.

Code Attachments

The attachment on the next page illustrates Ada code for the abstract data type Complex Number. It includes the two parts of a package: the ***public interface*** and the ***private implementation body***.

The two pages after that contain a package for the type Communication, which includes `Send` and `Receive` functions, and a buffer for loose-coupling.

The ***protected type*** construct of Ada controls concurrent access to shared data between tasks. Since it is a type definition, it is a template which must be instantiated with an object definition to create an actual instance. The effect of a protected type is to assure that only one task can execute changes to internal data structures (here, a buffer) at one time. This ensures consistent data management without the overhead of a third task. The coordination is achieved through the ***entry*** construct, which is a guarded function call. A ***guard*** allows only one active call at a time, other calls to the same entry are temporarily blocked until the controlling call is finished. The final page includes code for the Communications task, making it a concurrent procedure.

Language Generations (a recap)

Generation	*Exemplars*	*Characteristics*
0	pseudocodes	syntactic sugar for primitive assembly languages
1	FORTRAN	data and control correspond to machine architecture, linear, card-oriented
2	Algol	hierarchical name structures, block structure, strong typing, still linear and machine oriented
3	Pascal	simplicity and efficiency, user-defined data types, application-oriented
4	Ada	data abstraction, concurrency, still sequential, summary refinement of previous generations
5	LISP, Prolog, Smalltalk, JAVA	comprehensive, formal paradigms (functional, logical, object-oriented), self-documenting, recursive, provable, simulations, semantic constraints
6	?	hardware/software codesign, application specific, embedded, fine-grained strong parallelism, programming environments, language frameworks

The popular **C** language is a relatively unprincipled combination of first, second, and third generation language characteristics. The **C++** language modifies C to incorporate fourth and fifth generation characteristics, again without strongly embedded design principles.