Safe and Secure Software

An Invitation to **Ada 2005**

1

Safe Syntax

Courtesy of AdaCore
The GNAT Pro Company

John Barnes

Syntax is often considered to be a rather boring mechanical detail. The argument being that it is what you say that matters but not so much how it is said. That of course is not true. Being clear and unambiguous are important aids to any communication in a civilized world.

Similarly, a computer program is a communication between the writer and the reader, whether the reader be that awkward thing: the compiler, another team member, a reviewer or other human soul. Indeed, most communication regarding a program is between two people. Clear and unambiguous syntax is a great help in aiding communication and, as we shall see, avoids a number of common errors.

An important aspect of good syntax design is that it is a worthwhile goal to try to ensure that typical simple typing errors cause the program to become illegal and thus fail to compile, rather than having an unintended meaning. Of course it is hard to prevent the accidental typing of X rather than Y or + rather than * but many structural risks can be prevented. Note incidentally that it is best to avoid short identifiers for just this reason. If we have a financial program about rates and times then using identifiers R and T is risky since we could easily type the wrong identifier by mistake (the letters are next to each other on the keyboard). But if the identifiers are Rate and Time then inadvertently typing Tate or Rime will be caught by the compiler. This applies to any language of course.

Equality and assignment

It is obvious that assignment and equality are different things. If we do an assignment then we change the state of some variable. On the other hand, equality is simply an operation to test some state. Changing state and testing state are very different things and understanding the distinction is important.

Many programming languages have confused these fundamentally different logical operations.

In the earliest days of Fortran one wrote

```
X = X + 1
```

But this is really rather peculiar. In mathematics x never equals $x + 1$. What the Fortran statement means of course is "replace the current value of X by the old value plus one". But why misuse the equals sign in this way when society has been using the equals sign to mean equals for hundreds of years? (The equals sign dates from around 1550 when it was introduced by the English mathematician Robert Recorde.) The designers of Algol 60 recognized the problem and used the combination of a colon followed by an equals sign to mean assignment, thus

```
X := X + 1;
```

and this has the helpful consequence that the equals sign can unambiguously be used to mean equality, as in

```
if X = 0 then ...
```

The C language (like Fortran) adopted = for assignment and as a consequence C uses a double equals (==) to mean equality. This can cause much confusion.

Here is a fragment of a C program controlling the crossing gates on a railroad

```
if (the_signal == clear)
{
  open_gates( ... );
  start_train( ... );
}
```

The same program in Ada might be

```
if The_Signal = Clear then
  Open_Gates( ... );
  Start_Train( ... );
end if;
```

Now consider what happens if a programmer gets confused and accidentally forgets one of the equals signs in C thus

```
if (the_signal = clear)
{
  open_gates( ... );
  start_train( ... );
}
```

This still compiles but instead of just testing the_signal it actually assigns the value clear to the_signal. Moreover C unifies expressions (which have values) with assignments (which change state). So the assignment also acts as an expression and the result of the assignment is then used in the test. If the encoding is such that clear is not zero then the result will be true and so the gates are always opened, the_signal set to clear and the train started on its perilous journey. Conversely, if clear is encoded as zero, the test fails, the gates remain closed, and the train is blocked. In either case, things go badly wrong.

The pitfalls associated with the use of "=" for assignment and "==" for equality, and allowing assignments as expressions, are well known in the C community and have given rise to coding guidelines and analysis tools such as lint. However it is preferable for such pitfalls to be avoided in the first place, through appropriate language design and that is how Ada has approached this issue

If the Ada programmer were to accidentally use an assignment in the test

```
if The_Signal := Clear then                    -- illegal
```

then the program will simply fail to compile and all will be well.

Statement groups

It is often necessary to group a sequence of statements together – for example following a test using a keyword such as "if". There are two typical ways of doing this

- by bracketing the group of statements so that they act as one (as in C),
- by closing the sequence with something matching the "if" (as in Ada).

These are also illustrated by the railroad example. The statements to open the gates and to start the train both need to be obeyed if the condition is true.

In C we had

```
if (the_signal == clear)
{
  open_gates( ... );
  start_train( ... );
}
```

and now suppose we inadvertently add a semicolon at the end of the first line (easily done). The program becomes

```
if (the_signal == clear) ;
{
  open_gates( ... );
  start_train( ... );
}
```

We now find that the condition is governing the null statement which is implicitly present between the test and the newly inserted semicolon. We cannot see it because a null statement is just nothing. So no matter what the state of the signal, the gates are always opened and the train set going.

In Ada the corresponding error would result in

```
if The_Signal = Clear then ;                   -- illegal
  Open_Gates( ... );
  Start_Train( ... );
end if;
```

This is syntactically incorrect and so the error is safely caught by the compiler and the train wreck cannot occur.

Named notation

Another feature of Ada which is of a syntactic nature and can detect many unfortunate errors is the use of named associations in various situations. Dates provide a good illustration, because the order of the components varies according to local culture. Thus 12 January 2008 is written in Europe as 12/01/08 but in the US it is usually written as 01/12/08 (but not on the latest customs forms) whereas the ISO standard gives the year first, so would be 08/01/12.

In C we might declare a structure for manipulating dates as follows:

```
struct date {
  int day, month, year;
  } ;
```

which corresponds to the following type declaration in Ada

```
type Date is
  record
    Day, Month, Year: Integer;
  end record;
```

In C we might write

```
struct date today = {1, 12, 8};
```

But without looking at the type declaration we do not know whether this means 1 December 2008, 12 January 2008 or even 8 December 2001.

In Ada we have the option of writing

```
Today: Date := (Day => 1, Month => 12, Year => 08);
```

which uses named associations. Now it will be crystal clear if we ever write the values in the wrong order. (Note incidentally that Ada permits leading zeroes.).

We can also write the declaration as

```
Today: Date := (Month => 12, Day => 1, Year => 08);
```

which has the correct meaning and reveals the advantage that we do not need to remember the order in which the fields are declared.

Named associations can be used in other contexts in Ada as well. We might make similar errors with a function that has several parameters of the same type.

Suppose we have a function to compute the obesity index of a person. The two parameters are the height and the weight which could be given as floating point values in pounds and inches (or kilograms and centimeters if you are metric). So we might have in C:

```
float index(float height, float weight) {
  ...
  return ... ;
}
```

or in Ada

```
function Index(Height, Weight: Float) return Float is
  ...
  return ... ;
end;
```

Now in the case of the author, the appropriate call of the index function in C might be

```
my_index = index(68.0, 168.0);
```

But if by mistake the call were reversed

```
my_index = index(168.0, 68.0);
```

then we would have a very thin and very tall giant! (It's a curious coincidence that both values end in 68.0 as well.)

Such an unhealthy disaster can be avoided in Ada by using named parameter calls thus

```
My_Index := Index(Height => 68.0, Weight => 168.0);
```

Again we can give the parameters in whatever order we wish and no error will occur if we forget the order in the declaration of the function.

Named notation is a very valuable feature of Ada. Its use is optional but it is well worth using freely since not only does it help to prevent errors but it also makes the program easier to understand.