A brief introduction to

Ada 2012

by John Barnes

Chapter 2 - Expressions

Courtesy of AdaCore
The GNAT Pro Company

Rationale for Ada 2012: 2 Expressions

John Barnes
John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes the introduction of more flexible forms of expressions in Ada 2012.

There are four new forms of expressions. If expressions and case expressions define a value and closely resemble if statements and case statements. Quantified expressions take two forms using for all and for some to return a Boolean value. Finally, expression functions provide a simple means of parameterizing an expression without the formality of providing a function body.

These more flexible forms of expressions will be found invaluable in formulating contracts such as preconditions. It is interesting to note that Ada now has conditional expressions over 50 years after their introduction in Algol 60.

Keywords: rationale, Ada 2012.

1 Overview of changes

One of the key areas identified by the WG9 guidance document [1] as needing attention was improving the ability to write and enforce contracts. These were discussed in detail in the previous paper.

When defining the new aspects for preconditions, postconditions, type invariants and subtype predicates it became clear that without more flexible forms of expressions, many functions would need to be introduced because in all cases the aspect was given by an expression.

However, declaring a function and thus giving the detail of the condition, invariant or predicate in the function body makes the detail of the contract rather remote for the human reader. Information hiding is usually a good thing but in this case, it just introduces obscurity.

Four forms are introduced, namely, if expressions, case expressions, quantified expressions and expression functions. Together they give Ada some of the flexible feel of a functional language.

In addition, membership tests are generalized to allow greater flexibility which is particularly useful for subtype predicates.

The following Ada issues cover the key changes and are described in detail in this paper:

3 Qualified expressions and names

147 Conditional expressions

158 Generalizing membership tests

176 Quantified expressions

177 Expression functions

188 Case expressions

These changes can be grouped as follows.

First there are conditional expressions which come in two forms, if expressions and case expressions, which have a number of features in common (147, 188).

Then there is the introduction of quantified expressions which use **for all** to describe a universal quantifier and **for some** to describe an existential quantifier. Note that **some** is a new reserved word (176).

Next comes the fourth new form of expression which is the expression function (177).

Finally, membership tests are generalized (158) and there is a minor change regarding qualified expressions (3).

2 If expressions

It is perhaps very surprising that Ada does not have if expressions as well as if statements. In order to provide some background context we briefly look at two historic languages that are perhaps the main precursors to modern languages; these are Algol 60 [2] and CPL [3].

Algol 60 had conditional expressions of the form

```
Z := if X = Y then P else Q
```

which can be contrasted with the conditional statement

```
if X = Y then
  Z := P
else
  Z := Q
```

Conditional statements in Algol 60 allowed only a single statement in each branch, so if several were required then they had to be grouped into a compound statement thus

```
if X = Y then
  begin
    Z := P;  A := B
  end
else
  begin
    Z := Q;  A := C
  end
```

It may be recalled that statements were not terminated by semicolons in Algol 60 but separated by them. However, a null statement was simply nothing so the effect of adding an extra semicolon in some cases was harmless. However, accidentally writing

```
if X = Y then ;
  begin
    Z := P;  A := B
  end;
```

results in a disaster because the test then just covers a null statement and the assignments to Z and A always take place. The complexity of compound statements did not arise with conditional expressions.

The designers of Algol 68 [4] sensibly recognized the problem and introduced closing brackets thus

```
if X = Y then
  Z := P; A := B;
fi;
```

where **fi** matches the **if**. Conditional expressions in Algol 68 were similar

```
Z := if X = Y then P else Q fi;
```

An alternative shorthand notation was

```
Z := (X = Y | P | Q);
```

which was perhaps a bit too short.

The next major language in this series was Pascal [5]. The designers of Pascal rejected everything that had been learnt from Algol 68 and foolishly continued the Algol 60 style for compound

statements and also dropped conditional expressions. Only with Modula did they realise the need for bracketing rather than compounding.

The other foundation language was CPL [3]. Conditional statements in CPL took the following form

```
if X = Y then do Z := P
if X = Y then do § Z := P; A := B §
```

where compound statements were delimited by section symbols (note that the closing symbol has a vertical line through it).

From CPL came BCPL, B and C. Along the way, the expressive := for assignment got lost in favour of = which then meant that = had to be replaced by == for equality. And the section brackets became { and } so in C the above conditional statements become

```
if (X == Y) Z = P;
if (X == Y) {Z = P;  A = B;}
```

This suffers from the same stray semicolon problem mentioned above with reference to Algol 60.

Steelman [6] did not require Ada to have conditional expressions and since they were not required they were not provided (the requirements were treated with considerable reverence). A further influence might have been that the new language had to be based on one of Pascal, Algol 68 and PL/I and Ada is based on Pascal which did not have conditional expressions as mentioned above.

Moreover, the Ada designers felt that the Algol 68 style with reversed keywords such as **fi** (or worse **esac**) for conditional statements would not be acceptable to the USDoD or the public at large and so we have **end if** as the closing bracket thus

```
if X = Y then
  Z := P;
  A := B;
end if;
```

Remember that semicolons terminate statements in Ada and so those above are all required. Moreover, since null statements in Ada have to be given explicitly, placing a stray semicolon after **then** gives a compiler error.

The absence of conditional expressions is a pain. It leads to unnecessary duplication such as having to write

```
if X > 0 then
  P(A, B, D, E);
else
  P(A, C, D, E);
end if;
```

where all parameters but one are the same. This can even lead to disgusting coding using the fact that Boolean'Pos(True) is 1 whereas Boolean'Pos(False) is 0. Thus (assuming that B and C are of type Integer) the above could be written as a single procedure call thus

```
P(A, Boolean'Pos(X>0)*B+Boolean'Pos(X<=0)*C, D, E);
```

So it is a great relief in Ada 2012 to be able to write

```
P(A, (if X>0 then B else C), D, E);
```

A worse problem was when a static expression was required such as the initial value for a named number as in the following gruesome code

```
Febdays: constant := Boolean'Pos(Leap)*29 + Boolean'Pos(not Leap)*28;
```

which we can now thankfully write as

```
Febdays: constant := (if Leap then 29 else 28);
```

Note carefully that there is no **end if**. One reason is simply that it is logically unnecessary since there can be only a single expression after **else** and also **end if** would have been obtrusively heavy (compared say with **fi** of Algol 68). However, it was felt that some demarcation was required to aid clarity and so a conditional expression is always enclosed in parentheses. If the context already has parentheses then additional ones are not required. Thus in the case of a positional call with a single parameter we just write

```
P(if X > 0 then B else C);
```

but if we use named notation then extra parentheses are required

```
P(Para => (if X > 0 then B else C));
```

Note carefully that the term conditional expression in Ada 2012 embraces both if expressions and case expressions (which are discussed in the next section).

As expected, a series of tests can be done using **elsif** thus

```
P(if X > 0 then B elsif X < 0 then C else D);
```

and expressions can be nested

```
P(if X > 0 then (if Y > 0 then B else C) else D);
```

Without the rule requiring enclosing parentheses this could be written as

```
P(if X > 0 then if Y > 0 then B else C else D);        -- illegal
```

which seems more than a little confusing.

There is a special rule if the type of the expression is Boolean (that is of the predefined type Boolean or derived from it). In that case a final else part can be omitted and is taken to be true by default. Thus the following are equivalent

```
P(if C1 then C2 else True);
```

```
P(if C1 then C2);
```

Such abbreviations appear frequently in preconditions as was illustrated in the Introduction where we had

```
Pre => (if P1 > 0 then P2 > 0);
```

which has the obvious meaning that the precondition requires that if P1 is positive then P2 must be positive as well but if P1 is not positive then all is well and we don't care about P2.

This abbreviated form has the same effect as an implies operation.

```
R := C1 implies C2;          -- not Ada!
```

with the following truth table

	C1 = False	C1 = True
C2 = False	R = True	R = False
C2 = True	R = True	R = True

Some consideration was given to including such an operation in Ada 2012 (it existed in Algol 60). However, this is exactly the same as

```
R := not C1 or C2;
```

and so somewhat unnecessary. Moreover, although **implies** might appeal to some programmers it could lead to maintenance problems since it might be considered incomprehensible by many others.

There are important rules regarding the types of the various dependent expressions in the branches of an if expression. Basically they have to all be of the same type or convertible to the same expected type. But there are some interesting situations.

If the conditional expression is the argument of a type conversion then effectively the conversion is considered pushed down to the dependent expressions. Thus

```
X := Float(if P then A else B);
```

is equivalent to

```
X := (if P then Float(A) else Float(B));
```

As a consequence we can write

```
X := Float(if P then 27 else 0.3);
```

and it doesn't matter that 27 and 0.3 are not of the same type.

If the expected type is class wide, perhaps giving the initial value for a class wide variable V, then the individual dependent expressions have that same expected class wide type but they need not all be of the same specific type within the class. Thus we might write

```
V: Object'Class := (if B then A_Circle else A_Triangle);
```

where A_Circle and A_Triangle are objects of specific types Circle and Triangle which are themselves descended from the type Object.

If the expected type is a specific tagged type then various situations can arise regarding the various branches which are similar to the rules for calling a subprogram with several controlling operands. Either they all have to be dynamically tagged (that is class wide) or all have to be statically tagged. They might all be tag indeterminate in which case the conditional expression as a whole is also tag indeterminate.

Some obscure curiosities arise. Remember that the controlling condition for an if statement can be any Boolean type. Consider

```
type My_Boolean is new Boolean;

My_Cond: My_Boolean := ... ;

if (if K > 10 then X = Y else My_Cond) then          -- illegal
   ...
end if;
```

The problem here is that X = Y is of type Boolean but My_Cond is of type My_Boolean. Moreover, the expected type for the condition in the if statement is any Boolean type so it cannot make up its mind. We could overcome this foolishness by putting a type conversion around the if expression.

There are also rules regarding staticness. If all branches are static then a conditional expression as a whole is static as in the example of Febdays above. Thus the definition of a static expression has been extended to permit the inclusion of static conditional expressions.

The avid reader of the Reference Manual will find that the term *statically unevaluated* has been introduced. This relates to situations where expressions are not evaluated because a prior expression is static. Consider

```
X := (if B then P else Q);
```

If B, P and Q are all static then the conditional expression as a whole is static. If B is true then the answer is P and there is not any need to even look at Q. We say that Q is statically unevaluated and indeed it does not matter that if Q had been evaluated it would have raised an exception. Thus we might write

```
Average := (if Count = 0 then 0.0 else Total/Count);
```

and there is no risk of dividing by zero.

Similar rules regarding being statically unevaluated apply to short circuit conditions, case expressions, and membership tests.

As might be expected there are rules regarding access types and accessibility. The accessibility level of a conditional expression is simply that of the chosen dependent expression and thus (generally) determined dynamically.

Readers might feel that Ada has embarked on a slippery slope by introducing more flexibility thereby possibly damaging Ada's reputation for reliability. Certainly a number of additional rules have been required but from the users' point of view these are almost intuitive. It should be remembered that the difficulties in C stem from a combination of things

- that assignment is permitted as an expression,
- that integer values are used as Booleans,
- that null statements are invisible.

None of these applies to Ada so all is well.

3 Case expressions

Case expressions have much in common with if expressions and the two are collectively known as conditional expressions.

Thus given a variable D of the familiar type Day, we can assign the number of hours in a working day by

```
Hours := (case D is
                    when Mon .. Thurs => 8,
                    when Fri => 6,
                    when Sat | Sun => 0);
```

A slightly more adventurous example involving nested if expressions is

```
Days := (case M is
                    when September | April | June | November => 30,
                    when February =>
                            (if Year mod 100 = 0 then
                              (if Year mod 400 =0 then 29 else 28)
                                    else
                              (if Year mod 4 = 0 then 29 else 28)),
                    when others => 31);
```

The reader is invited to improve this!

Note the similarity to the rules for if expressions. There is no closing **end case**. Case expressions are always enclosed in parentheses but they can be omitted if the context already provides parentheses.

If M and Year are static then the case expression as a whole is also static. If M is static and equal to September, April, June or November then the value is statically known to be 30 so that the expression for February is statically unevaluated even if Year is not static. Note that the various choices are evaluated in order.

The rules regarding the types of the dependent expressions are exactly as for if expressions. Thus if the case expression is the argument of a type conversion then the conversion is effectively pushed down to the dependent expressions.

It is always worth emphasizing that an important advantage of case constructions is that they give a coverage check. Thus in the previous paper we had

```
subtype Pet is Animal
  with Static_Predicate =>
    (case Pet is
          when Cat | Dog | Horse => True,
          when Bear | Wolf => False);
```

which is much more reliable than

```
subtype Pet is Animal
  with Static_Predicate => Pet in Cat | Dog | Horse;
```

because when we add Rabbit to the type Animal, we are forced to include it in one branch of the case expression whereas it is all too easy to forget it using an if expression.

4 Quantified expressions

Another new form of expression in Ada 2012 is the quantified expression. The syntax is

```
quantified_expression ::=
          for quantifier loop_parameter_specification => predicate
        | for quantifier iterator_specification => predicate

quantifier ::= all | some

predicate ::= boolean_expression
```

The form involving iterator_specification concerns generalized iterators and will be found particularly useful with containers; it will be discussed in detail in a later paper. Here we will concentrate on the use of the familiar loop parameter specification.

The type of a quantified expression is Boolean. So we might write

```
B := (for all K in A'Range => A(K) = 0);
```

which assigns true to B if every component of the array A has value 0. We might also write

```
B := (for some K in A'Range => A(K) = 0);
```

which assigns true to B if some component of the array A has value 0.

Note that the loop parameter is almost inevitably used in the predicate. A quantified expression is very much like a for statement except that we evaluate the expression after => on each iteration rather than executing one or more statements. The iteration is somewhat implicit and the words **loop** and **end loop** do not appear.

The expression is evaluated for each iteration in the appropriate order (**reverse** can be inserted of course) and the iteration stops as soon as the value of the expression is determined. Thus in the case of **for all**, as soon as one value is found to be False, the overall expression is False whereas in the case of **for some** as soon as one value is found to be True, the overall expression is True. An iteration could raise an exception which would then be propagated in the usual way.

Like conditional expressions, a quantified expression is always enclosed in parentheses which can be omitted if the context already provides them, such as in a procedure call with a single positional parameter.

Incidentally, predicate is a fancy word meaning Boolean expression. Older folk might recall that it also means the part of a sentence after the subject. Thus in the sentence "I love Ada", the subject is "I" and the predicate is "love Ada".

The forms **for all** and **for some** are technically known as the universal quantifier and existential quantifier respectively.

Note that **some** is a new reserved word (the only one in Ada 2012). There were five new ones in Ada 95 (**aliased**, **protected**, **requeue**, **tagged** and **until**) and three new ones in Ada 2005 (**interface**, **overriding** and **synchronized**). Hopefully we are converging.

The type of a quantified expression can be any Boolean type (that is the predefined type Boolean or perhaps My_Boolean derived from Boolean). The predicate must be of the same type as the expression as a whole. Thus if the predicate is of type My_Boolean then the quantified expression is also of type My_Boolean.

The syntax for quantified expressions uses => to introduce the predicate. This is similar to the established notation in SPARK [7]. Consideration was given to using a vertical bar which is common in mathematics but that would have been confusing because of its use in membership tests and other situations with multiple choices.

As illustrated in the Introduction, quantified expressions will find their major uses in pre- and postconditions. Thus a procedure Sort on an array A of type Atype such as

```
type Atype is array (Index) of Float;
```

might have specification

```
procedure Sort(A: in out Atype)
  with Post => A'Length < 2 or else
                  (for all K in A'First .. Index'Pred(A'Last) => A(K) <= A(Index'Succ(K)));
```

where we are assuming that the index type need not be an integer type and so we have to use Succ and Pred. Note how the trivial cases of a null array or an array with a single component are dismissed first.

Quantified expressions can be nested. So we might check that all components of a two-dimensional array are zero by writing

```
B := (for all I in AA'Range(1) => (for all J in AA'Range(2) => AA(I, J) = 0));
```

This can be done rather more neatly using the syntactic form

```
for quantifier iterator_specification => predicate
```

as will be discussed in detail in a later paper. We just write

```
B := (for all E of AA => E = 0);
```

which iterates over all elements of the array AA however many dimensions it has.

5 Expression functions

The final new form to be discussed is the expression function. As outlined in the Introduction, an expression function provides the effect of a small function without the formality of introducing a body. It is important to appreciate that strictly speaking an expression function is basically another form of function and not another form of expression. But it is convenient to discuss expression functions in this paper because like conditional expressions and quantified expressions they arose for use with aspect clauses such as pre- and postconditions.

The syntax is

```
expression_function_declaration ::=
        [overriding_indicator]
        function_specification is
          (expression)
          [aspect_specification] ;
```

As an example we can reconsider the type Point and the function Is_At_Origin thus

```
package P is
  type Point is tagged
    record
      X, Y: Float := 0.0;
    end record;

  function Is_At_Origin(P: Point) return Boolean is
    (P.X = 0.0 and P.Y = 0.0)
      with Inline;

  ...
end P;
```

The expression function Is_At_Origin is a primitive operation of Point just as if it were a normal function with a body. If a type My_Point is derived from Point then Is_At_Origin would be inherited or could be overridden with a normal function or another expression function. Thus an expression function can be prefixed by an overriding indicator as indicated by the syntax.

Expression functions can have an aspect clause and since by their very nature they will be short, this will frequently be **with** Inline as in this example.

The result of an expression function is given by an expression in parentheses. The parentheses are included to immediately distinguish the structure from a normal body which could start with an arbitrary local declaration. The expression can be any expression having the required type. It could for example be a quantified expression as in the following

```
function Is_Zero(A: Atype) return Boolean is
  (for all J in A'Range => A(J) = 0);
```

This is another example of a situation where the quantified expression does not need to be enclosed in its own parentheses because the context supplied by the expression function provides parentheses.

Expression functions can be completions as well as standing alone and this introduces a number of possibilities. Remember that many declarations require completing. For example an incomplete type such as

```
type Cell;                          -- an incomplete type
```

is typically completed by a full type declaration later on

```
type Cell is
  record ... end record;          -- its completion
```

Completion also applies to subprograms. Typically the declaration (that is the specification plus semicolon) of a subprogram appears in a package specification thus

```
package P is
  function F(X: T);               -- declaration
  ...
end P;
```

and then the body of F which completes it appears in the body of P thus

```
package body P is
  function F(X: T) is             -- completion
  begin
    ...
  end F;
  ...
end P;
```

A function body cannot appear in a package specification. The only combinations are

function declaration F	function body F
in spec of P	in body of P
in body of P	in body of P
none	in body of P

Remember that mutual recursion may require that a body be given later so it is possible for a distinct declaration of F to appear in the body of P before the full body of F. In addition to the above the function body could be replaced by a stub and the proper body compiled separately but that is another story.

The rules regarding expression functions are rather different. An expression function can be declared alone as in the example of Is_At_Origin above; or it can be a completion of a function declaration and that completion can be in either the package specification or body. A frequently useful combination occurs with a private type where we need to make a function visible so that it can be used in a precondition and the expression function then occurs in the private part as a completion thus

```
package P is
  type Point is tagged private;
  function Is_At_Origin(P: Point) return Boolean
    with Inline;
  procedure Do_It(P: in Point; ... )
    with Pre => not Is_At_Origin;

private

  type Point is tagged
    record
      X, Y: Float := 0.0;
    end record;
```

```
function Is_At_Origin(P: Point) return Boolean is
  (P.X = 0.0 and P.Y = 0.0);

  ...
end P;
```

Note that we cannot give an aspect specification on an expression function used as a completion so it is given on the function specification; this makes it visible to the user. (This rule applies to all completions such as subprogram bodies and is not special to expression functions.)

An expression function can also be used in a package body as an abbreviation for

```
function Is_At_Origin(P: Point) return Boolean is
begin
  return P.X = 0.0 and P.Y = 0.0;
end Is_At_Origin;
```

The possible combinations regarding a function in a package are

function declaration F	expression function F
in spec of P	in spec or body of P
in body of P	in body of P
none	in spec or body of P

We perhaps naturally think of an expression function used as a completion to be in the private part of a package. But we could declare a function in the visible part of a package and then an expression function to complete it in the visible part as well. This is illustrated by the following interesting example of two mutually recursive functions.

```
package Hof is

  function M(K: Natural) return Natural;
  function F(K: Natural) return Natural;

  function M(K: Natural) return Natural is
    (if K = 0 then 0 else K – F(M(K–1)));

  function F(K: Natural) return Natural is
    (if K =0 then 1 else K – M(F(K–1)));

end Hof;
```

These are the Male and Female functions described by Hofstadter [8]. They are inextricably intertwined and both are given with completions for symmetry.

Almost inevitably, at least one of the expression functions in a mutually recursive pair will include an if expression (or else **or else**) otherwise the recursion will not stop.

Expression functions can also be declared in subprograms and blocks (they are basic declarative items). Moreover, an expression function that completes a function can also be declared in the subprogram or block.

This is illustrated by the following Gauss-Legendre algorithm which computes π to an amazing accuracy determined by the value of the constant K.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Long_Long_Float_Text_IO;
```

```
use Ada.Long_Long_Float_Text_IO;
with Ada.Numerics.Long_Long_Elementary_Functions;
use Ada.Numerics.Long_Long_Elementary_Functions;
procedure Compute_Pi is

  function B(N: Natural) return Long_Long_Float;

  function A(N: Natural) return Long_Long_Float is
    (if N = 0 then 1.0 else (A(N–1)+B(N–1))/2.0);

  function B(N: Natural) return Long_Long_Float is
   (if N = 0 then Sqrt(0.5) else Sqrt(A(N–1)*B(N–1)));

  function T(N: Natural) return Long_Long_Float is
    (if N = 0 then 0.25 else
       T(N–1)–2.0**(N–1)*(A(N–1)–A(N))**2);

  K: constant := 5;                    -- for example
  Pi: constant Long_Long_Float := ((A(K) + B(K))**2 / (4.0*T(K));
begin
  Put(Pi, Exp => 0);
  New_Line;
end Compute_Pi;
```

With luck this will output 3.14159265358979324 (depending on the accuracy of Long_Long_Float).

The functions A and B give successive arithmetic and geometric means. They call each other and so B is given as a function specification which is then completed by the expression function.

I am grateful to Brad Moore and to Ed Schonberg for these instructive examples.

The rules regarding null procedures (introduced in Ada 2005 primarily for use with interfaces) are modified in Ada 2012 to be uniform with those for expression functions regarding completion. Thus

```
procedure Nothing(X: in T) is null;
```

can be used alone as a declaration of a null operation for a type or as a shorthand for a traditional null procedure thus possibly completing the declaration

```
procedure Nothing(X: in T);
```

Expression functions and null procedures do not count as subprogram declarations and so cannot be declared at library level. Nor can they be used as proper bodies to complete stubs. Library subprograms are mainly intended for use as main subprograms and to use an expression function in that way would be somewhat undignified!

Thus if we wanted to declare a useful function to compute sin 2*x* from time to time, we cannot write

```
with Ada.Numerics.Elementary_Functions;
use Ada.Numerics.Elementary_Functions;
function Sin2(X: Float) is                    -- illegal
  (2.0 * Sin(X) * Cos(X));
```

but either have to write it out the long way or wrap the expression function in a package.

6 Membership tests

Membership tests in Ada 83 to Ada 2005 are somewhat restrictive. They take two forms

- to test whether a value is in a given range, or
- to test whether a value is in a given subtype.

Examples of these are

```
if M in June .. August then

if I in Index then
```

However, the restrictions are annoying. If we want to test whether it is safe to eat an oyster (there has to be an R in the month) then we would really like to write

```
if M in Jan .. April | Sep .. Dec then          -- illegal in Ada 2005
```

whereas we are forced to write something like

```
if M in Jan .. April or M in Sep .. Dec then
```

which means repeating M and then perhaps worrying about whether to use **or** or **or else**. Or in this case we could do the test the other way

```
if M not in May .. Aug then
```

What we would really like to do is use the vertical bar as in case statements and aggregates to select a combination of ranges, subtypes, and values.

Ada 2012 is much more flexible in this area. To see the differences it is probably easiest to look at the old and new syntax. The relevant old syntax for Ada 2005 is

```
relation ::=
        simple_expression [relational_operator simple_expression]
      | simple_expression [not] in range
      | simple_expression [not] in subtype_mark
```

where the last two productions define membership tests. The syntax regarding choices in aggregates and case statements in Ada 2005 is

```
discrete_choice_list ::= discrete_choice { | discrete_choice}

discrete_choice ::= expression | discrete_range | others

discrete_range ::= discrete_subtype_indication | range
```

The syntax in Ada 2012 is rather different and changes relation to use new productions for membership_choice_list and membership_choice (this enables the vertical bar to be used in membership tests). And then membership_test in turn uses choice_expression and choice_relation as follows

```
relation ::=
        simple_expression [relational_operator simple_expression]
      | simple_expression [not] in membership_choice_list

membership_choice_list ::= membership_choice { | membership_choice}

membership_choice ::= choice_expression | range | subtype_mark

choice_expression ::=
        choice_relation {and choice_relation}
      | choice_relation {or choice_relation}
      | choice_relation {xor choice_relation}
      | choice_relation {and then choice_relation}
      | choice_relation {or else choice_relation}

choice_relation ::= simple_expression [relational_operator simple_expression]
```

The difference between a choice_relation and a relation is that the choice_relation does not include membership tests. Moreover, discrete_choice is changed to

discrete_choice ::= choice_expression | *discrete*_subtype_indication | range | **others**

the difference being that a discrete_choice now uses a choice_expression rather than an expression as one of its possibilities.

The overall effect of the changes is to permit the vertical bar in membership tests without getting too confused by nesting membership tests.

Here are some examples that are now permitted in Ada 2012 but were not permitted in Ada 2005

```
if N in 6 | 28 | 496 then                    -- N is small and perfect!
if M in Spring | June | October .. December then
                                              -- combination of subtype, single value and
range
if X in 0.5 .. Z | 2.0*Z .. 10.0 then         -- not discrete or static
if Obj in Triangle | Circle then              -- with tagged types
if Letter in 'A' | 'E' | 'I' | 'O' | 'U' then -- characters
```

Membership tests are permitted for any type and values do not have to be static. There is no change here but it should be remembered that existing uses of the vertical bar in case statements and aggregates do require the type to be discrete and the values to be static.

Another important point about membership tests is that the membership choices are evaluated in order and as soon as one is found to be true (or false if **not** is present) then the relation as a whole is determined and the other membership choices are not evaluated. This is therefore the same as using short circuit forms such as **or else** and so gives another example of expressions which are statically unevaluated.

There is one very minor incompatibility. In Ada 2005 we can write

```
X: Boolean := ...
case X is
  when Y in 1 .. 10  => F(10);
  when others => F(5);
end case;
```

This is rather peculiar. The discrete choice Y **in** 1 .. 10 must be static. Suppose Y is 5, so that Y **in** 1 .. 10 is true; then if X is True, we call F(10) whereas if X is false we call F(5). And vice versa for values of Y not in the range 1 to 10.

This is syntactically illegal in Ada 2012 because a discrete choice can no longer be an expression and so be a membership test. This was imposed because otherwise we might have been tempted to write

```
X: Boolean := ...
case X is
  when Y in 1 .. 10 | 20 => F(10);
  when others => F(5);
end case;
```

and this is syntactically ambiguous because it might be parsed as (Y **in** 1 .. 10) | 20 rather than as if we were allowed to write Y **in** (1 .. 10) | 20. Although it would be rejected anyway because of the type mismatch.

A nastier example might make this clearer. Consider

```
case X is
  when Y in False | True => Do_This;
  when others => Do_That;
end case;
```

Now suppose that Y itself is of type Boolean. Is it (Y **in** False) | True rather than Y **in** (False | True)? If Y happens to have the value True then the first interpretation gives False | True so whatever the value of X we always Do_This but in the second interpretation we get just True so if X happens to be False we Do_That. So it really is seriously ambiguous without any type mismatch in sight and has to be forbidden.

However, this is clearly very unlikely to be a problem. Case statements over Boolean types are pretty rare anyway.

There is one other change to membership tests which concerns access types and so will be considered again in a later paper. The change is that membership tests can be used to check accessibility.

It is often the case that one uses a membership test before a conversion to ensure that the conversion will succeed. This avoids raising an exception which then has to be handled. Thus we might have

```
subtype Score is Integer range 1 .. 60;
Total: Integer;
S: Score;
...                                  -- compute Total somehow
if Total in Score then
  S := Score(Total)-- reliable conversion
  ...                          -- now use S knowing that it is OK
else
  ...                          -- Total was excessive
end if;
```

If we are indexing some arrays whose range is Score then it is an advantage to use S as an index since we know it will work and no checks are needed.

However, in Ada 2005, we cannot use a membership test to check accessibility. But Ada 2012 permits this and we can write

```
type Ptr is access all T;

procedure P(X: access T) is
  Local: Ptr;
begin
  if X in Ptr then
    Local := Ptr(X);           -- reliable conversion
    ...                        -- now use Ptr knowing that it is OK
  else
    ...                        -- would have failed accessibility check
  end if;
end P;
```

We could also do the check in a precondition thus

```
procedure P(X: access T)
  with Pre => X in Ptr;
```

Here we have a precondition where the expression is simply a membership test X **in** Ptr. Of course this does not avoid the exception because it will raise Assertion_Error if the accessibility is wrong.

7 Qualified expressions

We conclude this discussion of expressions by considering some points regarding names and primaries.

In Ada 2005 we have

```
name ::= direct_name | explicit_dereference | indexed_component
        | slice | selected_component | attribute_reference
        | type_conversion | function_call | character_literal

primary ::= numeric_literal | null | string_literal | aggregate | name
        | qualified_expression | allocator | (expression)
```

And in Ada 2012 we have

```
name ::= direct_name | explicit_dereference | indexed_component
        | slice | selected_component | attribute_reference
        | type_conversion | function_call | character_literal
        | qualified_expression | generalized_reference | generalized_indexing

primary ::= numeric_literal | null | string_literal | aggregate | name
        | allocator | (expression) | (conditional_expression) | (quantified_expression)
```

The important thing to observe here is that qualified_expression has moved from being a form of primary to being a name.

We also note the addition of conditional_expression and quantified_expression (both in parentheses) as forms of primary as discussed earlier in this paper and the addition of generalized_reference and generalized_indexing as forms of name. These are used in the new forms of iterator briefly alluded to at the end of the discussion on quantified expressions and which will be discussed in a later paper.

Returning to qualified expressions, the main reason for allowing them as names is to avoid unnecessary conversions as mentioned in the Introduction.

Consider

```
A: T;                                   -- object of type T
type Art is array (1 .. 10) of T;       -- array of type T
function F(X: Integer) return Art;
```

A function call can be used as a prefix and so a call returning an array can be indexed as in

```
A := F(3)(7);
```

which assigns to A the value of the 7th component of the array returned by the call of F.

Now suppose that F is overloaded so that F(3) is ambiguous. The normal solution to such ambiguities is to use qualification and write Art'(F(3)) as in

```
A := Art'(F(3))(7);                     -- illegal in Ada 2005
```

but this is illegal in Ada 2005 because a qualified expression is not a name and so cannot be used as a prefix. What one has to do in Ada 2005 is either copy the wretched array (really naughty) or add a type conversion (a type conversion *is* a name) thus

```
A := Art(Art'(F(3)))(7);
```

This is really gruesome; but in Ada 2012, qualification is permitted as a name so we can simply write

```
A := Art'(F(3))(7);                    -- OK in Ada 2012
```

Although a qualified expression is now classed as a name rather than a primary, a qualified variable is not considered to be a variable. As a consequence, a qualified variable cannot be used as the destination of an assignment or as an actual parameter corresponding to an **out** or **in out** parameter. This would have added complexity for no useful purpose. Ambiguity generally involves calls on overloaded functions, and the result of a function call is always a constant, so ambiguous names of variables are unlikely!

Other uses might involve strings which can also give rise to ambiguities. For example

```
("a string")'Length
```

is ambiguous (it could be a String or Wide_String). But now we can write

```
String'("a string")'Length
```

which was not permitted in Ada 2005.

References

[1] ISO/IEC JTC1/SC22/WG9 N498 (2009) Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of Amendment 2 to ISO/IEC 8652.

[2] P. Naur (ed.) *Revised Report on the Algorithmic Language ALGOL 60* (1963) Communications of the Association for Computing Machinery, Vol. 6, p. 1.

[3] D. W. Barron et al (1963) *The main features of CPL*, Computer Journal vol. 6, pp 134-143.

[4] A. van Wijngaarden et al (eds) (1973) *Revised Report on the Algorithmic Language – ALGOL 68*, Springer-Verlag.

[5] K. Jensen and N. Wirth (1975) *Pascal User Manual and Report*, Springer-Verlag.

[6] Defense Advanced Research Projects Agency (1978) *Department of Defense Requirements for High Order Computer Programming Languages – STEELMAN*, USDoD.

[7] J. G. P. Barnes (2003) *High Integrity Software, The SPARK Approach to Safety and Security*, Addison-Wesley.

[8] D. R. Hofstadter (1980) *Gödel, Escher, Bach: an Eternal Golden Braid*, Basic Books.