Ada/SPARK 2014 – Mini Cheat Sheet

Packages

Specification (*.ads)

```
package P with SPARK_Mode is
   procedure Something (X : Integer);
end P;
```

Body (*.adb)

```
package body P with SPARK_Mode is
   procedure Something (X : Integer) is
   begin
      -- ..
   end Something;
end P;
```

Referencing Packages

```
with P; -- import content of package P
use P;  -- make content of P usable w/o prefix "P."
```

Subprograms

With return value

```
function F1 (X : Integer) return Integer is
   var : constant Integer := X + 1;
begin
   return var;
end f1;
-- same in short ("expression function"):
function F1 (X : Integer) return Integer is (X + 1);
```

No return value

```
procedure p1 (Y : in out Natural) is
begin
   Y := F1 (Y);
   Put_Line ("Y is now" & Natural'Image(Y));
end p1;
```

Types

Predefined Types

```
Boolean, Integer, Natural, Positive,
Float, Character, Duration, String
```

Creating New Types

```
-- type compatible to predefined Integer:
subtype Months is Integer range 1 .. 12;
-- completely new, Float-incompatible type:
type Bitcoin is new Float;
-- type that wraps around:
type Hours is mod 24; -- ranges from 0 to 23
```

Array Types

```
-- declare type
type Arr_T is array (positive range <>) of Integer;
-- create array variable
A : Arr_T (1 .. 2) := (2, 3);
```

Composite Types

```
type My_Vector is record
  x : Float;
  y : Float;
end record;
```

Enumeration Types

```
type My_Weekdays is (Monday, Holiday, Friday);
```

Conditional Control Flow

```
if A then -- ...
elsif B then -- ..
else -- ...
end if;
```

```
case weekday is
   when Monday | Friday =>
      Do_Work;
   when others =>
      null;
end case;
```

Loops

Counting Loop

```
for i in Integer range 1 .. 10 loop
   -- ...
end loop;
```

Iterator Loop

```
for i in My_Weekdays'Range loop
   -- see first column, also works with other types
end loop;
```

Head-Controlled

```
while A > 5 loop
   -- ...
end loop;
```

Body-Controlled

```
My_Loop : loop
   A := Calc; -- subprogramm call
   exit My_Loop when A > 5;
end loop My_Loop;
```

Attributes

S for subtype, A for array. Some of them also work on the instance.

```
S'First       -- lowest value in range of S
S'Last        -- highest value in range of S
A'First       -- first index of array
A'Last        -- last index of array
A'Length      -- length of array
S'Image(v)    -- stringification of value in v
S'Range       -- iterator in loops over type range
A'Range       -- iterator in loops over array indizes
S'Size        -- size in bits of instantiated object
S'Succ(v)     -- value that follows v in type range
S'Pred(v)     -- value that preceded v in type range
S'Val(x)      -- value of type whose position = x
S'Pos(x)      -- position of value x in the type S
S'Floor(x)    -- largest integral value ≤ x in S
S'Ceiling(x)  -- smallest integral value ≥ x in S
```

Operators

```
and, or, xor, not -- Logical operators
+, -, *, /, mod, rem, **, abs, =, /=, <, <=, >, >=
```

Boolean Short-Circuit Operators

```
if A and then B then ... -- only check B when A true
if A or else B then ... -- only check B when A false
```

Subprogram Contracts

Preconditions

```
procedure p (X, Y : Integer)
  with Pre => Y /= 0 and then X > 0;
```

Postconditions

```
function Increment (X : Integer) return Integer
  with Pre  => X < Integer'Last,
       Post => Increment'Result = X + 1;
```

```
procedure Increment (X : in out Integer)
  with Pre  => X < Integer'Last,
       Post => X = X'Old + 1;
```

Global Variables

```
procedure P
  with Global => (Input  => (A, B),
                  Output => (C, D),
                  In_Out => (E));
--  may read A, B and E; and write C, D, E
```

Information Flow

```
procedure Sum (A, B : Integer; Result : out Integer)
  with Depends => (Result => (A, B));
--  Result *must* depend on A and B
```

Loop (In)Variants

```
pragma Loop_Invariant (J in Low .. High);
pragma Loop_Variant (Increases => i,
                     Decreases => x);
```

Testing and Proof

Assertions

```
pragma Assert (X >= 0); -- abort execution for
      negative values
pragma Assert (X >= 0); -- can never fail because of
       previous assert
```

Assumptions

```
procedure No_Contract (Y : Integer) is
begin
  pragma Assume (Y >= 0);
  -- now analysis only considers positive values
  pragma Assert (Y >= 0); -- never fails in analysis
end No_Contract;
```

Suppressing False Warnings (Only use with utmost care!)

```
return A / B;
pragma Annotate (GNATprove, False_Positive,
                 "divide by zero",
                 "reviewed by John Doe");
```

```
X : Integer;
pragma Annotate (GNATprove, Intentional,
                 """X"" is not initialized",
                 "reviewed by John Doe");
```

Created by Martin Becker, https://github.com/mbeckersys
Revision 2017-12-06 12:18:52 +0100