**SPARK 2014**

Quick Reference Examples

### Aspects

```plaintext
aspectsSpecification ::=  
  with aspectMark [=> aspectDefinition]  
  , aspectMark [=> aspectDefinition] 
```

### SPARK Mode

```plaintext
package P  
  with SPARK_Mode => On  
is  
  -- package spec is SPARK, so can be used  
  -- by SPARK clients  
end P;  
package body P  
  with SPARK_Mode => Off  
is  
  -- body is NOT SPARK, so assumed to  
  -- be full Ada  
end P;  
```

### Subprogram Contracts

#### PRECONDITIONS

```plaintext
function F (X : Integer) return Integer  
with Pre => X * X < 100;  
```

```plaintext
procedure P (X : Integer; Y : Integer)  
with Pre => X + Y = 0;  
```

```plaintext
procedure Some_Call  
with Pre => Initialized;  
-- before it is declared  
Initialized : Boolean := False;  
```

#### POSTCONDITIONS

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

### CONTRACT CASES

```plaintext
procedure Bounded_Add  
((X, Y : in Integer; Z : out Integer)  
with Contract_Cases  
((X + Y in Integer'Range) => Z = X + Y,  
  Integer'First < X + Y => Z = Integer'First,  
  X + Y > Integer'Last => Z = Integer'Last);  
```

### GLOBAL CONTRACTS

```plaintext
procedure P  
with Global => (Input => (A, B, C),  
Output => (I, M, N),  
In_Out => (X, Y, Z),  
Proof_In => (I, J, K));  
```

### DEPENDS CONTRACTS

Contracts for information flow analysis.

```plaintext
procedure Sum  
(A, B : in Integer; Result : out Integer)  
with Depends => (Result => (A, B));  
```

#### ASSUME

No verification condition generated - soundness alert!

Use with great care.

```plaintext
pragma Assume (Ticks < Type_Time'Last);  
```
### Package Contracts

#### ABSTRACT STATE

```plaintext
package P
  with Abstract_State =>
    (Essential_State, Result_Cache)
  -- Parentheses not required
  -- if only one state abstraction
is
end P;
```

#### REFINED STATE

```plaintext
package body P
  with Refined_State =>
    (Essential_State => (E1, E2),
     Result_Cache => Cache)
is
end P;
```

#### INITIALIZATION

```plaintext
package A_Stack
  with Abstract_State =>
    Stack,
  Initializes =>
    Stack,
  Initial_Condition =>
    Stack_Empty
  -- state abstractions are not listed
  -- in an Initializes contract if they are
  -- not initialized by package elaboration
is
  function Stack_Empty return Boolean
    with Global => Stack;
end A_Stack;
```

#### EXTERNAL STATE

```plaintext
package Output_Port
  with System.Storage_Elements;
  is
  Sensor : Integer
    with Volatile,
    Async_Readers;
  Address =>
    System.Storage_Elements.To_Address (16#ACECAFE#);
end Output_Port;
```

### PART_OF

```plaintext
package P
  with Abstract_State => State_P
is
private
  Hidden_Var : Integer with
    Part_Of => State_P;
  ...
end P;
```

```plaintext
package Q
  with Abstract_State => (S1, S2),
  Initializes => S1
is
private
  package Q.Child
    with Abstract_State =>
      (Child_State with
       Part_Of => Q.S1),
  Initializes => Child_State
is
end Q.Child;
```

### Warnings and Check Message Control

```plaintext
package body Warnings_Example is
  pragma Warnings (Off, "formal parameter "X" is not referenced");
  procedure Mumble (X : Integer) is
    pragma Warnings (On, "formal parameter "X" is not referenced");
    -- X is ignored here, because ... etc.
  begin
    null;
  end Mumble;
end Warnings_Example;
```

Remember that every failed check message corresponds to a soundness issue and should be reviewed / justified individually.

```plaintext
return (X + Y) / (X - Y);
pragma Annotate
  (GNATprove, False_Positive,
   "divide by zero", "reviewed by John Smith");
```

```plaintext
procedure Do_Something (X, Y : in out Integer) with
  Depends => ((X, Y) => (X, Y));
pragma Annotate
  (GNATprove, Intentional,
   "incorrect dependency "Y => X"",
    "Dependency is kept for compatibility reasons");
```

### Flags for the SPARK Tools

Options for the compiler and GNATprove.

#### OVERFLOW CHECKING MODES

GNAT Pro compiler switch controls semantics of overflow checks in assertions (contracts) and code. Three modes:

1 = strict Ada semantics for overflow checking
2 = minimized overflow checking
3 = eliminated - no possibility of overflow (mathematical semantics)

Example: `.gnato13`
- First digit specifies overflow mode for code.
- Second digit specifies overflow mode for contracts.