SPARK

- Elementy języka SPARK (6h)
- Pakiety (3h)
- Sprawdzanie zależności (2h)
- Dowodzenie (3h)

Przykład

Tokeneer

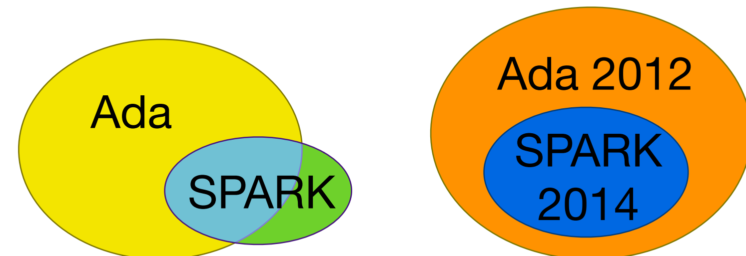
- 65 plików źródłowych
- 36167 wierszy kodu
- 3057 warunków do sprawdzenia i udowodnienia
- 2 minuty weryfikacji (Intel Core 2 Duo 2.8GHz, level=0)
- 0 ostrzeżeń
- 20 niezmienników pętli

```
if Success and then
   (RawDuration * 10 <= Integer (DurationT'Last) and
    RawDuration * 10 >= Integer (DurationT'First)) then
  Value := DurationT (RawDuration * 10);
else
  Success := False;
end if;
```

medium: overflow check might fail

```
if Success and then
   (RawDuration <= Integer (DurationT'Last) / 10 and
    RawDuration >= Integer (DurationT'First) / 10) then
  Value := DurationT (RawDuration * 10);
else
  Success := False;
end if;
```

Elementy języka SPARK



a) SPARK 83, SPARK 95 i SPARK 2005

b) SPARK 2014

Poniższe własności języka Ada 2012 nie są obecnie dostępne w SPARK:

- aliasy nazw
- wskaźniki i dynamiczna alokacja pamięci
- instrukcja `goto`
- wyrażenia i funkcje z efektami ubocznymi
- obsługa wyjątków
- typy sterowane (tworzenie, przypisania, niszczenie)
- zadania/wielowątkowość[1)]

1) można dowodzić własności zadań w uproszczonym modelu **Ravenscar**

Struktury sterowania

Instrukcja **if**:

```
if A < 0 then
  A := -A;
  D := 1;
end if;

if A in 1 .. 12 then
  B := 17;
end if;

if A > B then
  E := 1;
  F := A;
else
  E := 2;
  F := B;
end if;
```

```
if A = B then
  F := 3;
elsif A > B then
  F := 4;
else
  F := 5;
end if;

if A >= B and A >= C then
  G := 6;
elsif B > A and B > C then
  G := 7;
elsif C > A and C > B then
  G := 8;
end if;
```

Operator równości:

- = równe
- /= różne

Operatory relacji:

- < mniejsze niż
- <= mniejsze lub równe
- > większe
- >= większe lub równe

Operatory logiczne:

- **`not`** logiczna negacja
- **`and`** logiczna koniunkcja
- **`or`** logiczna alternatywa
- **`xor`** alternatywa wykluczająca
- **`and then`** leniwa koniunkcja
- **`or else`** leniwa alternatywa

Operatory przynależenia:

- **`in`** należy
- **`not in`** nie należy

Instrukcja **case**:

```
Success := True;
case Ch is
  when 'a' .. 'z' =>
    H := 1;
  when 'A' .. 'Z' =>
    H := 2;
  when '0' .. '9' =>
    H := 3;
  when '.' | '!' | '?' =>
    H := 4;
  when others =>
    H := 5;
    Success := False;
end case;
```

Warunkowe wyrażenia **if**:

```
if A > B then              C := (if A > B then D + 5 else F / 2);
  C := D + 5;
else
  C := F / 2;
end if;

if A > B then              C := (if A > B then D + 5 elsif A = B then 2 * A
  C := D + 5;                    else F / 2);
elsif A = B then
  C := 2 * A;
else
  C := F / 2;
end if;

if X >= 0.0 then           Y := Sqrt (if X >= 0.0 then X else -2.0 * X);
  Y := Sqrt (X);
else
  Y := Sqrt (-2.0 * X);
end if;
```

Koniecznie w nawiasach i bez **end if**.

Jeśli wyrażenie warunkowe ma wartość logiczną, to część **else** może być pominięta:

(if C - D = 0 then E > 2 else True)   (if C - D = 0 then E > 2)

Logiczne wyrażenia warunkowe **if** często są stosowane w warunkach pre, post i inwariantach pętli:

Pre => (if A < 0 then B < 0);

Warunkowe wyrażenie **case**:

```
Value := (case Letter is
             when  'A' | 'E' | 'I' | 'L' | 'U' |
                   'N' | 'O' | 'R' | 'S' | 'T'    => 1,
             when  'D' | 'G'                      => 2,
             when  'B' | 'C' | 'M' | 'P'          => 3,
             when  'F' | 'H' | 'V' | 'W' | 'Y'    => 4,
             when  'K'                            => 5,
             when  'J' | 'X'                      => 8,
             when  'Q' | 'Z'                      => 10);
```

Logiczne wyrażenia warunkowe **case** często występują w warunkach pre, post i niezmiennikach pętli.

Instrukcja pętli:

```
loop
  ADC.Read (Temperature);                         -- Read the temperature from the ADC
  Calculate_Valve (Current_Temp => Temperature,   -- Calculate the new
                   New_Setting => Valve_Setting); --   gas valve setting
  DAC.Write (Valve_Setting);                      -- Change the gas valve setting
end loop;

Sum := 0;
loop
  Ada.Integer_Text_IO.Get (Value);
  exit when Value < 0;
  Sum := Sum + Value;
end loop;

Approx := X / 2.0;
while abs (X - Approx ** 2) > Tolerance loop
  Approx := 0.5 * (Approx + X / Approx);
end loop;
```

```
for Count in Integer range 5 .. 8 loop
  Ada.Integer_Text_IO.Put (Count);
end loop;

for Count in reverse Integer range 5 .. 8 loop
  Ada.Integer_Text_IO.Put (Count);
end loop;

A := 9;
B := 2;

for Count in Integer range A .. B loop    -- With a null range, this
  Ada.Integer_Text_IO.Put (Count);        -- loop iterates zero times
end loop;

for Count in reverse Integer range A .. B loop    -- With a null range, this
  Ada.Integer_Text_IO.Put (Count);                -- loop iterates zero times
end loop;
```

```
  Value    : Integer;
  Modified : Boolean;

begin -- procedure Example
  Ada.Text_IO.Put_Line ("Enter a number:");
  Ada.Integer_Text_IO.Get (Value);
  Bounded_Increment (Bound => Limit / 2,
                     Value => Value,
                     Changed => Modified);
  if Modified then
    Ada.Text_IO.Put_Line ("Your number was changed to ");
    Ada.Integer_Text_IO.Put (Item => Value,
                             Width => 1);
  end if;
end Example;
```

Podprogramy

```
with Ada.Text_IO;
with Ada.Integer_Text_IO;
procedure Example is

  Limit : constant Integer := 1_000;

  procedure Bounded_Increment
    (Value   : in out Integer; -- A value to increment
     Bound   : in     Integer; -- The maximum that Value may take
     Changed : out    Boolean) -- Did Value change?
  is
  begin
    if Value < Bound then
      Value := Value + 1;
      Changed := True;
    else
      Changed := False;
    end if;
  end Bounded_Increment;
```

Tryby parametrów:

- **in** Używa się do przekazywania danych od wywoływacza do procedury. W procedurze parametr w trybie in traktowany jest jako stała. Argumentem wywołania może być dowolne wyrażenie, którego wartość odpowiada typowi formalnego parametru.

- **out** Używa się do przekazania danych z procedury do wywoływacza. Powinno się uważać formalny parametr w procedurze jako niezainicjowana zmienna. Argument wywołania musi być zmienną, której typ jest zgodny z typem parametru formalnego.

- **in out** Używa się do modyfikowania argumentu wywołania. Wartość jest przekazywana do procedury, modyfikowana przez nią i zwracana do wywoływacza. Argument musi być zmienną.

Zasięg identyfikatora określa się następującymi dwiema regułami:

- Zasięg identyfikatora zawiera wszystkie instrukcje następujące po jego definicji, wewnątrz regionu zawierającego definicję (region ten to jednostka programu taka jak pakiet, procedura albo funkcja). Zawiera on wszystkie zagnieżdżone regiony za wyjątkiem tych, których dotyczy druga reguła.

- Zasięg identyfikatora nie rozszerza się do zagnieżdżonego regionu, jeśli zdefiniowano w nim homograf. Regułę tę nazywa się *regułą przesłaniania* (ang. name precedence).

homograf
wyraz pisany identycznie jak inny wyraz, ale różniący się od niego wymową, znaczeniem i pochodzeniem; homogram

Funkcje w SPARK mogą mieć tylko parametry w trybie **in**. Mogą one korzystać ze zmiennych globalnych ale nie mogą ich zmieniać. Te ograniczenia zapewniają, że funkcje nie mają efektów ubocznych.

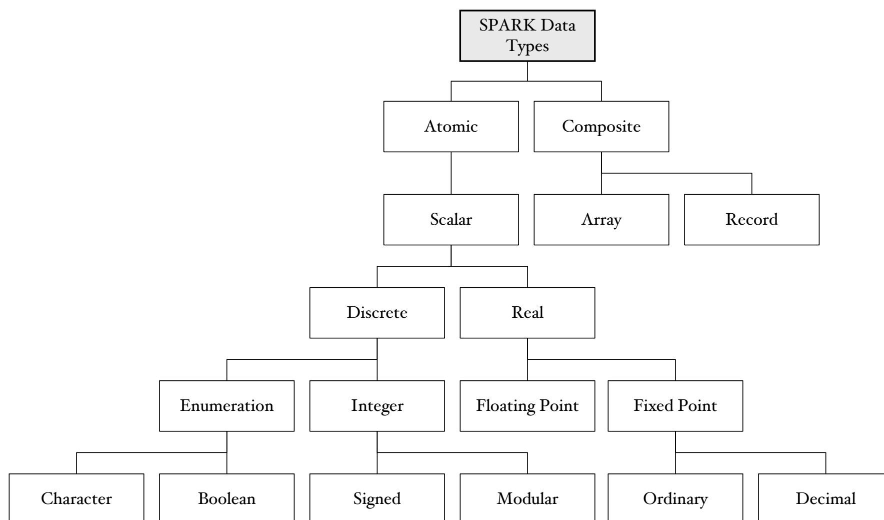
```
function Sqrt (X : in Float; Tolerance : in Float) return Float is
  Approx : Float; -- An approximation of the square root of X
begin
  Approx := X / 2.0;
  while abs (X - Approx ** 2) > Tolerance loop
    Approx := 0.5 * (Approx + X / Approx);
  end loop;
  return Approx;
end Sqrt;
```

Funkcje zapisane w postaci jednego wyrażenia:

```
function Double (First : in Integer;
                 Second : in Integer) return Boolean is
  (abs Second = 2 * abs First);
```

Wyrażenie musi być w nawiasie.

Typy danych



Typy skalarne

- Typ skalarny jest typem atomowym z dodatkową własnością uporządkowania. Wartości mogą być porównywane operatorami relacji (<, <=, > i >=).

- Są dwa rodzaje typów skalarnych: liczby rzeczywiste i typ dyskretny, który ma dodatkową własność jednoznacznego następnika i poprzednika.

Liczby rzeczywiste

- Liczby rzeczywiste nie mogą być reprezentowane dokładnie w pamięci komputera.
- Liczby które mają dokładną reprezentację nazywa się **liczbami modelowymi**.
- Każda liczba inna niż modelowa jest przybliżona i reprezentowana najbliższą liczbą modelową.
- Błąd reprezentacji danej liczby jest równy różnicy między nią a modelową liczbą ją reprezentującą.
- Dla przykładu w przedziale od 10,000.0 do 10,001.0 tylko 1,024 liczby reprezentowane są dokładnie w reprezentacji IEEE 754 pojedynczej precyzji.

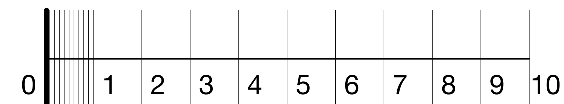
Liczby rzeczywiste zmiennopozycyjne

```
with Ada.Text_IO;
procedure Good_Types is
  -- Declarations of two floating point types
  type Feet   is digits 4 range 0.0 .. 100.0; co najmniej 4 cyfry dziesiętne w mantysie
  type Inches is digits 3 range 0.0 .. 12.0;  co najmniej 3 cyfry dziesiętne w mantysie

  -- Instantiation of input/output packages for feet and inches
  package Feet_IO is new Ada.Text_IO.Float_IO (Feet);
  package Inch_IO is new Ada.Text_IO.Float_IO (Inches);

  function To_Feet (Item : in Inches) return Feet is
  begin
    return Feet (Item) / 12.0;
  end To_Feet;

  Room_Length    : Feet;
  Wall_Thickness : Inches;
  Total          : Feet;

begin
  Feet_IO.Get (Room_Length);
  Inch_IO.Get (Wall_Thickness);
  Total := Room_Length + 2.0 * To_Feet (Wall_Thickness);
  Feet_IO.Put (Item => Total, Fore => 1, Aft => 1, Exp => 0);
end Good_Types;
```

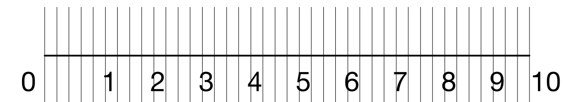
Liczby rzeczywiste stałopozycyjne

- W większości języków programowania dostępne są tylko liczby zmiennopozycyjne.
- Liczby zmiennopozycyjne reprezentują szerszy zakres wartości.
- Arytmetyka na liczby stałopozycyjne wykonywana jest standardowymi instrukcjami maszynowymi na liczbach całkowitych (dużo większa szybkość niż zmiennopozycyjne). Wiele tanich wbudowanych mikroprocesorów, mikrokontrolerów i cyfrowych procesorów sygnałów nie wspiera arytmetyki zmiennopozycyjnej.
- W liczbach stałopozycyjnych maksymalny błąd reprezentacji jest stały w całym zakresie, natomiast w reprezentacji zmiennopozycyjnej rośnie wraz ze wzrostem wartości.

Rozkład liczb modelowych



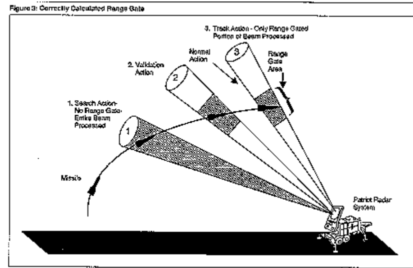
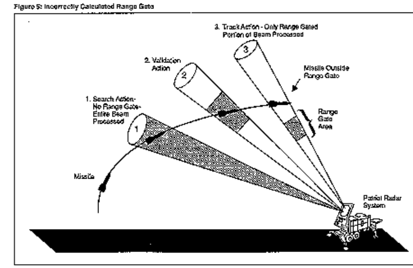
a) liczby rzeczywiste zmiennopozycyjne (digits = 1)



b) liczby rzeczywiste stałopozycyjne (delta = 0.25)

Przykład

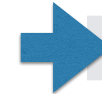
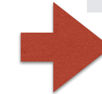
25.02.1991 Nieudane przechwycenie rakiety Scud przez baterię Patriot podczas operacji Pustynna Burza (śmierć 28 amerykańskich żołnierzy).

Ilustracje z raportu GAO/IMTEC-92-26 Patriot Missile Software Problem.

Złe działanie systemu spowodowane było reprezentacją czasu w postaci 24-bitowych liczb zmiennopozycyjnych, w której rozkład liczb modelowych skupiony jest w dolnym zakresie wartości.

Przykład cd.

Hours	Seconds	Absolute inaccuracy (Seconds)	Approximate shift in missile range gate
0	0.0	0.0000	0
1	3600.0	0.0034	7
8	28800.0	0.0275	55
20	72000.0	0.0687	137
48	172800.0	0.1648	330
72	259200.0	0.2472	494
100	360000.0	0.3433	687

Deklarując typ stałopozycyjny specyfikujemy maksymalny odstęp między liczbami modelowymi jaki możemy zaakceptować (maksymalny błąd reprezentacji jest wówczas połową tej odległości):

```
type Thirds is delta 1.0 / 3.0 range 0.0 .. 100_000.0;
type Volts  is delta 2.0**(-12) range 0.0 .. 5.0;
```

Chociaż w typie Thirds zadano odstęp 1/3, to faktycznie będzie on równy 1/4 (najwyższa potęga 2 nieprzekraczająca wartości 1/3). Z tego powodu typy takie są nazywane **binarnymi typami stałopozycyjnymi**.

W **dziesiętnym typie stałopozycyjnym** zadany odstęp jest potęgą 10 oraz zadaje się łączną liczbę cyfr dziesiętnych:

```
type Dollars is delta 0.01 digits 12;
```

Operacje wejścia/wyjścia można wykonywać dzięki rodzajowym pakietom **Ada.Text_IO.Fixed_IO** i **Ada.Text_IO.Decimal_IO**:

```
package Thirds_IO  is new Ada.Text_IO.Fixed_IO (Thirds);
package Volts_IO   is new Ada.Text_IO.Fixed_IO (Volts);
package Dollars_IO is new Ada.Text_IO.Decimal_IO (Dollars);
```

Typy dyskretne

- typy wyliczeniowe
- typy całkowite
  - całkowite ze znakiem
  - całkowite bez znaku (arytmetyka modulo)

```
function Next_Day (Day : in Day_Type) return Day_Type is
begin
    if Day = Day_Type'Last then
        return Day_Type'First;
    else
        return Day_Type'Succ(Day);
    end if;
end Next_Day;
```

Typy wyliczeniowe

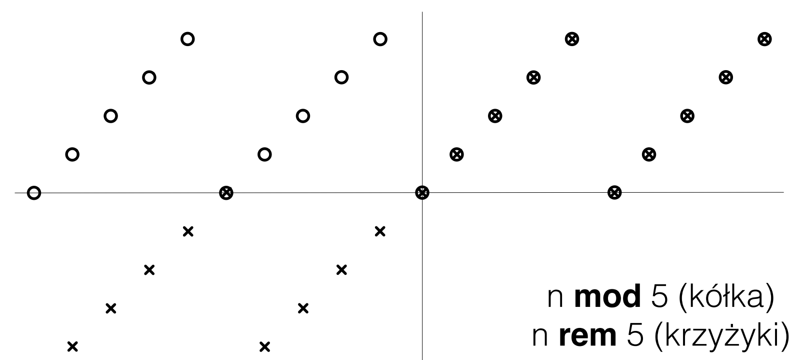
```
type Day_Type is (Monday, Tuesday, Wednesday, Thursday,
                  Friday, Saturday, Sunday);
type Traffic_Light_Color is (Red, Green, Yellow);
type Pixel_Color is (Red, Green, Blue, Cyan,
                     Magenta, Yellow, Black, White);
```

Atrybut	Opis
’First	Zwraca dolny kres typu
’Last	Zwraca górny kres typu
’Image	Zwraca łańcuch równoważny danej wartości
’Value	Zwraca wartość równoważną danemu łańcuchowi
’Succ	Zwraca następnik danej wartości
’Pred	Zwraca poprzednik danej wartości

Całkowite ze znakiem

```
type Pome      is range   0 .. 120;
type Citrus    is range -17 .. 30;
type Big_Range is range -20 .. 1_000_000_000_000_000_000;  -- trylion
```

Atrybuty ’First, ’Last, ’Succ, ’Pred i ’Image oraz operacje +, -, *, /, **, **abs**, **rem** i **mod**.



n	n mod 3 (Ada)	n rem 3 (Ada)	n % 3 (C)
-3	0	0	0
-2	1	-2	-2
-1	2	-1	-1
0	0	0	0
1	1	1	1
2	2	2	2
3	0	0	0

```
package Pome_IO   is new Ada.Text_IO.Integer_IO (Pome);
package Citrus_IO is new Ada.Text_IO.Integer_IO (Citrus);
package Big_IO    is new Ada.Text_IO.Integer_IO (Big_Range);
```

Całkowite bez znaku (arytmetyka modulo)

```
type Digit  is mod 10;    -- range is from 0 to 9
type Byte   is mod 256;   -- range is from 0 to 255
type Nybble is mod 16;    -- range is from 0 to 15
type Word   is mod 2**32; -- range is from 0 to 4,294,967,295

Value : Nybble;

Value := 12 + 8;  -- Value is assigned 4
```

Dodatkowe operatory logiczne **and**, **or** i **xor** działają na bitowych reprezentacjach (bit po bicie).

Logiczny operator **not** daje różnicę między górnym ograniczeniem typu a wartością, co dla ograniczenia będącego potęgą dwójki pokrywa się z negacją bit po bicie.

Podtypy

```
subtype Uppercase    is Character range 'A' .. 'Z';
subtype Negative     is Integer range Integer'First .. -1;

type Day_Type        is (Monday, Tuesday, Wednesday, Thursday,
                         Friday, Saturday, Sunday);
subtype Weekday      is Day_Type range Monday .. Friday;
subtype Weekend      is Day_Type range Saturday .. Sunday;

type Pounds          is digits 6 range 0.0 .. 1.0E+16;
subtype UPS_Weight   is Pounds range 1.0 .. 100.0;
subtype FedEx_Weight is Pounds range 0.1 .. 1.0;

Total    : Pounds;
Box      : UPS_Weight;
Envelope : FedEx_Weight;

-- Adding two different subtypes with same base type
Total := Box + Envelope;
```

Typy skalarne a dowodzenie

Definiowanie własnych typów i podtypów może ułatwić dowodzenie poprawności programów.

A := B / C;

Jeśli zadeklarujemy C jako typu, który nie zawiera zera, to SPARK nie będzie musiał dowodzić, że C nigdy nie przyjmie wartości zero.

A := (B + C) / 2;

Jeśli B i C będą dużymi wartościami, to ich suma może przekroczyć zakres wartości mieszczących się w akumulatorze procesora. Deklarując typy lub podtypy z ograniczonymi zakresami dla B i C dowód jest dużo prostszy dla programu dowodzącego.

Typy tablicowe

```
type Index_Type is range 1 .. 1000;
type Inventory_Array is array (Index_Type) of Natural;

subtype Lowercase is Character range 'a' .. 'z';
type Percent is range 0 .. 100;
type Frequency_Array is array (Lowercase) of Percent;

type Day is (Monday, Tuesday, Wednesday, Thursday,
             Friday, Saturday, Sunday);
type On_Call_Array is array (Day_Type) of Boolean;

Inventory : Inventory_Array;
Control   : Frequency_Array;
Unknown   : Frequency_Array;
On_Call   : On_Call_Array;
```

```
Inventory(5)     := 1_234;
Control ('a')    := 2 * Control('a');
On_Call (Sunday) := False;

Total_Days := 0;
for Day in Day_Type loop
    if On_Call (Day) then
        Total_Day := Total_Day + 1;
    end if;
end loop;

Unknown := Control; -- makes the array Unknown a copy of
                    -- the array Control

if Control = Unknown then
    Put("The values in the two arrays are identical");
elsif Control < Unknown then
    Put("The values in Control come lexicographically before those in Unknown");
end if;
```

Plasterki tablic

```
-- Copy elements 11-20 into locations 1-10
Inventory (1 .. 10) := Inventory (11 .. 20);

-- Copy elements 2-11 into location 1-10
Inventory (1 .. 10) := Inventory (2 .. 11);
```

Chociaż indeksowanie i wycinanie plasterków dostają się do elementów tablicy to wyniki są różne.

Wyrażenie Inventory (5) jest liczbą naturalną a Inventory (5 .. 5) jest tablicą złożoną z jednej liczby naturalnej.

Tablice wielowymiarowe

```
type Row_Index is range 1 .. 1000;
type Col_Index is range -5 .. 5;

type Two_D_Array is array (Row_Index, Col_Index) of Float;

type One_Row is array (Col_Index) od Float;
type Another_2D_Array is array (Row_Index) of One_Row;

Canary : Two_D_Array; -- A 2-dimensional array variable
Finch : Another_2D_Array; -- An array of arrays variable

-- Assign zero to row 12, column 2 of the 2-D array
Canary (12, 2) := 0.0;

-- Asign zero to the second component of the 12th array
Finch (12)(2) := 0.0;
```

Plasterki można tylko wycinać tylko z tablic jednowymiarowych (np. z Finch ale nie z Canary).

Tablicowe typy ograniczone i nieograniczone

```
type Float_Array is array (Positive range <>) of Float;

Illegal : Float_Array; -- The declaration will not compile

subtype Small_Array is Float_Array (1 .. 10);
subtype Large_Array is Float_Array (1000 .. 9999);

Small : Small_Array; -- An array of 10 Float values
Large : Large_Array; -- An array of 9,000 Float values

Large (1001 .. 1010) := Small; -- Copy 10 values
if Small /= Large (2001 .. 2010) then -- Compare 10 values
    -- Copy 21 values
    Large (2001 .. 2021) := Small & 14.2 & Small; -- concatenation
end if;
```

Przykład

Specyfikacja podprogramu z parametrem będącym nieograniczoną tablicą:

```
function Average (Values : in Float_Array) return Float;
```

Przykłady wywołań:

```
Avg := Average (Small);                -- Average of 10 values
Avg := Average (Large);                -- Average of 9,000 values
Avg := Average (Large (2001 .. 2010)); -- Average of 10 values
Avg := Average (Large & Small);        -- Average of 9,010 values
```

Atrybuty tablic

Atrybut	Opis
’First	Zwraca dolne ograniczenie indeksu
’Last	Zwraca górne ograniczenie indeksu
’Length	Zwraca liczbę elementów w tablicy
’Range	Zwraca zakres indeksu (czyli ’First .. ’Last)

Przykład cd.

Treść funkcji `Average`, w której użyto atrybutów tablicy:

```
function Average (Values : Float_Array) return Float is
    Sum : Float;
begin
    Sum := 0.0;
    for Index in Values'Range loop
        Sum := Sum + Values (Index);
    end loop;
    return Sum / Float (Values'Length);
end Average;
```

Agregaty tablicowe

```
Small := (0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 4.0, 3.0, 2.0, 1.0); -- by position
Small := (1 .. 10 => 0.0);                                    -- by name
Small := (1 .. 5 => 0.0, 6 .. 10 => 1.0);
Small := (1 | 3 | 5 | 7 | 9 => 0.0, others => 1.0);
Small := (others => A + Sqrt (B));

    type Row_Range is range -1 .. 1;
    type Col_Range is range 1 .. 4;
    type Table_Array is array (Row_Range, Col_Range) of Integer;

    Table : Table_Array; -- 2-dimensional array with 12 elements

    -- Assign all elements by position
    Table := ((1, 2, 3, 4),
              (5, 6, 7, 8),
              (9, 8, 7, 6));

    -- Assign rows by name, columns by position
    Table := (-1 => (1, 2, 3, 4),
               0 => (5, 6, 7, 8),
               1 => (9, 8, 7, 6));
    -- Assign all elements by name
    Table := (-1 => (1 .. 4 => 2),
               0 => (1 .. 3 => 3, 4 => 6),
               1 => (2 => 5, others => 7));
    Table := (-1 .. 1 => (1 .. 4 => 0));
    Table := (others => (others => 0));
```

Przykład

```
with Ada.Text_IO;
procedure Palindrome is

  function Is_Palindrome (Item : in String) return Boolean is
    Left_Index  : Natural;
    Right_Index : Natural;
  begin
    Left_Index := Item'First;
    Right_Index := Item'Last;
    loop
      exit when Left_Index >= Right_Index or else
                Item (Right_Index) /= Item (Left_Index);
      Left_Index := Left_Index + 1;
      Right_Index := Right_Index - 1;
    end loop;
    return Left_Index >= Right_Index;
  end Is_Palindrome;
```

Łańcuchy znaków

```
-- Type String is defined in Ada.Standard as
type String is array (Positive range <>) of Character;

subtype Name_String is String (1 .. 20); --- A constrained array type

Currency : constant String := "Dollars";

Name : Name_String; -- A string containing 20 characters
Address : String (1 .. 40); -- A string containing 40 characters
City : String (3 .. 22); -- A string containing 20 characters

City := Address; -- Illegal Assigning 40 characters to a 20 character string
Name := "Peter"; -- Illegal Assigning 5 characters to a 20 character string
Address := Name; -- Illegal Assigning 20 characters to a 40 character string

City := Name;          -- Legal Both source abd target contain 20 characters
Name := Address (1 .. 20); -- Legal Both source abd target contain
                           -- 20 characters
Address := Name & Name; -- Legal Both source abd target contain
                        -- 40 characters
Address (9 .. 28) := Name; -- Legal Both source abd target contain
                           -- 20 characters

Count := 5;
Name (1 .. Count) := "Peter";
Put (Name (1 .. Count));  -- Display Peter
```

Przykład cd.

```
  max_Length : constant Positive := 100;
  subtype Line_Type is String (1 .. Max_Length);

  Line : Line_Type;
  Count : Natural;

begin
  Ada.Text_IO.Put_Line ("Enter a line");
  Ada.Text_IO.Get (Item => Line, Last => Count);
  if Is_Palindrome (Line (1 .. Count)) then
    Ada.Text_IO.Put_Line ("is a palindrome");
  else
    Ada.Text_IO.Put_Line ("is not a palindrome");
  end if;
end Palindrome;
```

liczba przeczytanych znaków

Typy rekordowe

```
subtype Part_ID is Integer range 1000 .. 9999;
type Dollars is delta 0.01 digits 7 range 0.0 .. 10_000.0;

type Part_Rec is
  record
    ID : Part_ID;
    Price : Dollars;
    Quantity : Natural;
  end record;

Part     : Part_Rec;
Discount : Dollars;

Part.ID       := 1234;
Part.Price    := 1_856.25;
Part.Quantity := 597;
Discount      := 0.15 * Part.Price;
```

Agregaty rekordowe

```
Part := (1234, 1_856.25, 597); -- Assign values by position

Part := (ID       => 1234,  -- Assign values by name
         Quantity => 597,
         Price    => 1_856.25);
```

Wyróżniki

```
type Part_Array is array (Positive range <>) of Part_Rec;

type Inventory_List (Max_Size : Positive) is
  record
    Size : Natural;
    Items : Part_Array (1 .. Max_Size);
  end record;

Inventory : Inventory_List (Max_Size => 1000);

-- Append a new part to inventory list
if Inventory.Size = Inventory.Max_Size then
  Put_Line ("The inventory list is full");
else
  Inventory.Size := Inventory.Size + 1;
  Inventory.Items (Inventory.Size) := New_Part;
end if;
```

Wyróżniki są polami rekordu, przy czym muszą być typu dyskretnego.

Typy pochodne

```
-- Define a floating point type
type Gallons is digits 6 range 0.0 .. 100.0;

-- Define a new type derived from Gallons
type Imperial_Gallons is new Gallons;
```

- Imperial_Gallons jest typem pochodnym a Gallons jest jego typem rodzicielskim.
- Typy pochodne nie muszą ograniczać się do skalarów.
- Typy pochodne mogą być tworzone z tablic i rekordów.
- Dziedzina pochodnego typu jest kopią dziedziny typu rodzicielskiego.
- Typ pochodny i jego typ rodzicielski są różnymi typami.
- Głównie typy pochodne stosuje się w programowaniu obiektowym ale wykracza to poza język SPARK.

Przeciążanie podprogramów

```
procedure Calc (A : in  Integer;
                C : out Integer);

procedure Calc (A : in  Integer;
                C : out Float);

procedure Calc (A : in  Integer;
                B : in  Integer;
                C : out Integer);
```

- Możemy mieć dwa podprogramy o tej samej nazwie dopóki różnią się one sygnaturą.
- Sygnaturę podprogramu tworzy liczba, typy i porządek parametrów (również typ wartości zwracanej przez funkcję).
- Nazwy parametrów i ich tryby nie są częścią sygnatury.

Podprogramy rodzajowe (ang. *generic subprograms*)

Przykład

```
function Count (Source : in String;
                Pattern : in Character) return Natural is
-- Returns the number of times Pattern occurs in Source
  Result : Natural := 0;
begin
  for Index in Source'Range loop
    if Source (Index) = Pattern then
      Result := Result + 1;
    end if;
  end loop;
  return Result;
end Count;
```

Czy dla zliczania innych obiektów w innych tablicach trzeba pisać osobne funkcje?

Przykład

```
type Point is
  record
    X_Coord : Float;
    Y_Coord : Float;
  end record;
P1 : Point;
P2 : Point;

function "<=" (Left : in Point; Right : in Point) return Boolean is
-- Returns True when Left is the same distance from
-- or closer to the origin than Right
  Left_Squared  : Float;
  Right_Squared : Float;
begin
  Left_Squared  := Left.X_Coord ** 2 + Left.Y_Coord ** 2;
  Right_Squared := Right.X_Coord ** 2 + Right.Y_Coord ** 2;
  return Left_Squared <= Right_Squared; -- Calls <= for Float numbers
end "<=";

-- Function called as infix operator
if P1 <= P2 then
  Put_Line ("P1 is not further from origin than P2");
else
  Put_Line ("P1 is further from origin than P2");
end if;

-- "Normal" function call
if "<=" (P1, P2) then ...
```

Formalny typ rodzajowy	Dopuszczalne typy faktyczne
type T **is range** <>;	Dowolny typ całkowity ze znakiem.
type T **is mod** <>;	Dowolny typ całkowity bez znaku.
type T **is** (<>);	Dowolny typ dyskretny.
type T **is digits** <>;	Dowolny typ zmiennopozycyjny.
type T **is delta** <>;	Dowolny zwykły typ stałopozycyjny.
type T **is delta** <> **digits** <>;	Dowolny dziesiętny typ stałopozycyjny.
type T **is array** (Indx) **of** Cmp;	Dowolna tablica z typem indeksu Indx i typem elementów Cmp. Formalne i faktyczne parametry tablicy muszą być jednocześnie ograniczone albo nieograniczone.
type T **is private**;	Dowolny typ, dla którego przypisanie i relacja równości są dostępne (bez ograniczeń).
type T **is limited private**;	Dowolny typ.

Przykład cd.

Specyfikacja:

```
generic
  type Component_Type is private;
  type Index_Type is (<>);
  type Array_Type is array (Index_Type range <>) of Component_Type;
function Generic_Count (Source : in Array_Type;
                        Pattern : in Component_Type) return Natural;
```

Implementacja:

```
function Generic_Count (Source : in Array_Type;
                        Pattern : in Component_Type) return Natural is
  Result : Natural := 0;
begin
  for Index in Source'Range loop
    if Source (Index) = Pattern then
      Result := Result + 1;
    end if;
  end loop;
  return Result;
end Generic_Count;
```

Przykład cd.

```
type Percent is range 0 .. 100;
type Percent_Array is array (Character range <>) of Percent;

function Percent_Count is new Generic_Count
                               (Component_Type => Percent,
                                Index_Type     => Character,
                                Array_Type     => Percent_Array);

The_Count := Percent_Count(Source  => (5, 6, 7, 5, 3, 4, 19, 16, 5, 23),
                           Pattern => 5);

function Char_Count is new Generic_Count
                            (Component_Type => Character,
                             Index_Type     => Positive,
                             Array_String   => String);

The_Count := Char_Count (Source  => "How now brown cow",
                         Pattern => "w");
```

Pakiety

- Jednostka biblioteczna to osobno kompilowana jednostka programu.
- W Adzie jednostką biblioteczną może być
  - podprogram
  - pakiet
  - jednostka rodzajowa (ang. *generic unit*)

Pakiety definiujące

- Pakiety definiujące grupują razem powiązane ze sobą stałe i typy.
- Wykorzystywane przez współpracujących programistów pracujących nad różnymi częściami dużego programu.

Przykład

```ada
with Ada.Numerics;
package Common_Units is

  type Degrees is digits 18 range 0.0 .. 360.0;
  type Radians is digits 18 range 0.0 .. 2.0 * Ada.Numerics.Pi;

  type Volts is delta 1.0 / 2.0**12 range -45_000.0 .. 45_000.0;
  type Amps  is delta 1.0 / 2.0**16 range -1_000.0 .. 1_000.0;
  type Ohms  is delta 0.125         range 0.0 .. 1.0E8;

  type Light_Years is digits 12 range 0.0 .. 20.0E9;

  subtype Percent is Integer range 0 .. 100;
end Common_Units;
```

- Definicje sześciu typów i jednego podtypu.
- Wykorzystanie stałej π z pakietu Ada.Numerics.
- Brak podprogramów zatem nie będzie części implementującej (próba kompilacji treści dla tej specyfikacji skończy się błędem).

Przykład cd.

```ada
with Common_Units; use type Common_Units.Ohms;
with Ada.Text_IO; use Ada.Text_IO;
procedure Ohms_Law is

  package Ohm_IO  is new Fixed.IO (Common_Units.Ohms);
  package Amp_IO  is new Fixed.IO (Common_Units.Amps);
  package Volt_IO is new Fixed.IO (Common_Units.Volts);

  A  : Common_Units.Amps;
  R1 : Common_Units.Ohms;
  R2 : Common_Units.Ohms;
  V  : Common_Units.Volts;
begin
  Put_Line("Enter current and two resistances");
  Amp_IO.Get (A);
  Ohm_IO.Get (R1);
  Ohm_IO.Get (R2);
  V := A * (R1 + R2);
  Put ("The voltage drop over the two resistors is ");
  Volt_IO.Put (Item => V,
               Fore => 1,
               Aft => 2,
               Exp => 0);
  Put_Line (" volts");
end Ohms_Law;
```

prefiks przed nazwami typów

można korzystać z operatora bez prefiksu

Przykład (Ada 2012)

```ada
with System.Dim.Mks; use System.Dim.Mks;
with System.Dim.Mks_IO; use System.Dim.Mks_IO;
with Text_IO; use Text_IO;
procedure Free_Fall is
  G : constant Mks_Type := 9.81 * m / (s**2);
  T : Time := 10.0 * s;
  Distance : Length;
begin
  Put ("Gravitational constant: ");
  Put (G, Aft => 2, Exp => 0);
  Put_Line ("");
  Distance := 0.5 * G * T;   -- błąd!!!
  Put ("distance traveled in 10 seconds of free fall: ");
  Put (Distance, Aft => 2, Exp => 0);
  Put_Line ("");
end Free_Fall;
```

```
gcc -c free_fall.adb
free_fall.adb:12:13: dimensions mismatch in assignment
free_fall.adb:12:13: left-hand side has dimension [L]
free_fall.adb:12:13: right-hand side has dimension [L.T**(-1)]
```

Pakiety narzędziowe

- Pakiet narzędziowy grupuje razem stałe, typy, podtypy i podprogramy potrzebne do realizacji konkretnego serwisu.
- Pakiet **Ada.Numerics.Elementary_Functions** jest pakietem narzędziowym, który zawiera 29 funkcji matematycznych jak pierwiastek, funkcje trygonometryczne i logarytmy dla wartości Float.
- Jest również rodzajowa wersja pakietu **Ada.Numerics.Elementary_Functions**, która umożliwia jego konkretyzację dla dowolnego typu zmiennopozycyjnego.

Przykład

```
package Display_Control is

   procedure Bold_On;
   -- Everything sent to the screen after this procedure
   -- is called will be displayed in bold characters

   procedure Blink_On;
   -- Everything sent to the screen after this procedure
   -- is called will be blinking

   procedure Normal;
   -- Everything sent to the screen after this procedure
   -- is called will be displayed normally

end Display_Control;
```

Przykład cd.

```
with Ada.Text_IO;
with Ada.Characters.Latin_1;  -- Characters in the
                              -- ISO 8859-1 character set
package body Display_Control is

-- Assumes that the display accepts and processes American
-- National Standards Institute (ANSI) escape sequences.

   -- Code to start an ANSI control string (the Escape
   -- control character and the left bracket character)
   ANSI_Start : constant String :=
                         Ada.Characters.Latin_1.ESC & '[';

   procedure Bold_On is
   begin -- "ESC[1m" turns on Bold
      Ada.Text_IO.Put (ANSI_Start & "1m");
      -- Send any buffered characters to the display
      Ada.Text_IO.Flush;
   end Bold_On;

   procedure Blink_On is
   begin -- "ESC[5m" turns on Blink
      Ada.Text_IO.Put (ANSI_Start & "5m");
      Ada.Text_IO.Flush;
   end Blink_On;

   procedure Normal is
   begin -- "ESC[0m" turns off all attributes
      Ada.Text_IO.Put (ANSI_Start & "0m");
      Ada.Text_IO.Flush;
   end Normal;

end Display_Control;
```

Pakiety definiujące typy

- Pakiety definiujące typy tworzą **abstrakcyjne typy danych** (ang. *abstract data types* ADTs).
- Abstrakcyjny typ danych składa się ze zbioru wartości i operacji, które są niezależne od żadnej konkretnej implementacji.

Przykład

```
package Bounded_Queue_V1 is
-- Version 1, details of the queue type are not hidden

   subtype Element_Type is Integer;

   type Queue_Array is array (Positive range <>) of Element_Type;
   type Queue_Type (Max_Size : Positive) is
      record
        Count : Natural;     -- Number of items
        Front : Positive;    -- Index of first item
        Rear  : Positive;    -- Index of last item
        Items : Queue_Array (1 .. Max_Size);   -- The element array
      end record;

   function Full (Queue : in Queue_Type) return Boolean;

   function Empty (Queue : in Queue_Type) return Boolean;

   function Size (Queue : in Queue_Type) return Natural;

   function First_Element (Queue : in Queue_Type) return Element_Type
      with
          Pre => not Empty (Queue);
```

Przykład cd.

```
   function Last_Element (Queue : in Queue_Type) return Element_Type
      with
         Pre => not Empty (Queue);

   procedure Clear (Queue : in out Queue_Type)
      with
         Post => Empty (Queue) and then Size (Queue) = 0;

   procedure Enqueue (Queue : in out Queue_Type;
                      Item  : in     Element_Type)
      with
         Pre  => not Full (Queue),
         Post => not Empty (Queue) and then
                 Size (Queue) = Size (Queue'Old) + 1 and then
                 Last_Element (Queue) = Item;

   procedure Dequeue (Queue : in out Queue_Type;
                      Item  :    out Element_Type)
      with
         Pre  => not Empty (Queue),
         Post => Item = First_Element (Queue'Old) and then
                 Size (Queue) = Size (Queue'Old) - 1;

end Bounded_Queue_V1;
```

Przykład

```
with Bounded_Queue_V1;  use Bounded_Queue_V1;
with Ada.Text_IO;       use Ada.Text_IO;
procedure Bounded_Queue_Example_V1 is
-- Uses the first version of the bounded queue package

   My_Queue : Bounded_Queue_V1.Queue_Type (Max_Size => 100);
   Value    : Integer;

begin
   Clear (My_Queue);  -- Initialize queue
   for Count in Integer range 17 .. 52 loop
      Enqueue (Queue => My_Queue, Item => Count);
   end loop;
   for Count in Integer range 1 .. 5 loop
      Dequeue (Queue => My_Queue, Item => Value);
      Put_Line (Integer'Image (Value));
   end loop;
   Clear (My_Queue);
   Value := Size (My_Queue);
   Put_Line ("Size of cleared queue is " & Integer'Image (Value));
end Bounded_Queue_Example_V1;
```

Przykład

```
package body Bounded_Queue_V1 is

   function Full (Queue : in Queue_Type) return Boolean is
   begin
      return Queue.Count = Queue.Max_Size;
   end Full;

   function Empty (Queue : in Queue_Type) return Boolean is
   begin
      return Queue.Count = 0;
   end Empty;

   function Size (Queue : in Queue_Type) return Natural is
   begin
      return Queue.Count;
   end Size;

   function First_Element (Queue : in Queue_Type) return Element_Type is
   begin
      return Queue.Items (Queue.Front);
   end First_Element;

   function Last_Element (Queue : in Queue_Type) return Element_Type is
   begin
      return Queue.Items (Queue.Rear);
   end Last_Element;
```

Przykład cd.

```
   procedure Clear (Queue : in out Queue_Type) is
   begin
      Queue.Count := 0;
      Queue.Front := 1;
      Queue.Rear  := Queue.Max_Size;
   end Clear;

   procedure Enqueue (Queue : in out Queue_Type;
                      Item  : in     Element_Type) is
   begin
      Queue.Rear := Queue.Rear rem Queue.Max_Size + 1;
      Queue.Items (Queue.Rear) := Item;
      Queue.Count := Queue.Count + 1;
   end Enqueue;

   procedure Dequeue (Queue : in out Queue_Type;
                      Item  :    out Element_Type) is
   begin

      Item := Queue.Items (Queue.Front);
      Queue.Front := Queue.Front rem Queue.Max_Size + 1;
      Queue.Count := Queue.Count - 1;
   end Dequeue;

end Bounded_Queue_V1;
```

Wprowadzenie do kontraktów

- Kontrakty z przykładu pakietu dla kolejki są podane w postaci **aspektów**.
- Aspekt opisuje własność jednostki.
- Ada 2012 definiuje blisko 70 różnych aspektów jakie mogą być użyte w programie.
- Więcej o aspektach będzie w części poświęconej kontraktom.

- Specyfikacja typowego aspektu składa się z nazwy, strzałki => i definicji.

```
procedure Clear (Queue : in out Queue_Type)
   with
      Post => Empty (Queue) and then Size (Queue) = 0;
```

- Nazwa **`Post`** wskazuje, że aspekt jest warunkiem końcowym (ang. *postcondition*) dla procedury.
- Ten aspekt wymaga boolowskiej definicji po strzałce.
- W boolowskiej definicji mogą być odwołania do funkcji zdefiniowanych w pakiecie (np. **`Empty`** i **`Size`**).
- Oczekuje się, że warunek końcowy będzie spełniony po wykonaniu operacji **`Clear`**, tj. kolejka będzie pusta i miała rozmiar równy zeru.
- W implementacji może być błąd, zatem konieczne jest zweryfikowanie czy **`Clear`** spełnia jej warunek końcowy.

- Oczywistym sposobem weryfikacji jest testowanie:
  - napisać program testujący
  - wstawić w nim elementy do kolejki
  - wywołać operację **`Clear`**
  - sprawdzić wartości zwracane przez **`Empty`** i czy **`Size`**
- Ada 2012 umożliwia dodanie warunku końcowego w postaci asercji na końcu procedury (opcja **`-gnata`** kompilatora gcc).
- Sprawdzanie warunku końcowego na koniec każdego wywołania zwiększa czas obliczeń.
- Powodzenie testów nie gwarantuje, że warunek końcowy będzie spełniony przy każdym możliwym wykonaniu.
- Można użyć narzędzia **GNATprove** do formalnego udowodnienia warunku końcowego bez wykonywania kodu.

analiza statyczna

```
procedure Enqueue (Queue : in out Queue_Type;
                   Item  : in     Element_Type)
   with
      Pre  => not Full (Queue),
      Post => not Empty (Queue) and then
              Size (Queue) = Size (Queue'Old) + 1 and then
              Last_Element (Queue) = Item;
```

- Nazwa aspektu **`Pre`** wskazuje warunek wstępny dla podprogramu.
- Zdefiniowane boolowskie wyrażenie powinno być spełnione za każdym wywołaniem podprogramu.
- Może zawierać wywołania funkcji (np. **`Full`**).
- Procedura **`Enqueue`** nie powinna być wywoływana gdy kolejka jest już pełna.
- Opcją kompilatora **`-gnata`** można dodawać warunki wstępne jako asercje sprawdzane przy każdym wywołaniu podprogramu.
- Można użyć **GNATprove** do statycznej weryfikacji, że każde wywołanie podprogramu spełnia warunek wstępny.

- Atrybut `'Old` odnosi się do oryginalnej wartości parametru w trybie **in out**.
- Warunek wstępny odwołuje się do stanu parametru przed wywołaniem, zatem atrybut `'Old` nie jest potrzebny.
- Kolejka w warunku wstępnym not Full (Queue) odnosi się do kolejki przekazywanej do modyfikacji przez procedurę.
- Atrybut `'Result` odnosi się do wartości zwracanej przez funkcję:

```
function Sqrt (Item : in Natural) return Natural
  with
    Post => Sqrt'Result ** 2 <= Item and then
            (Sqrt'Result + 1) ** 2 > Item;
```

Ukrywanie informacji

- Szczegóły definicji typu `Queue_Type` w pakiecie `Bounded_Queue_V1` były publiczne.
- Programiści korzystający z tego pakietu mogą zignorować zdefiniowane w nim operacje i dostawać się do składowych rekordu manipulując kolejką na własną rękę.
- Takie manipulowanie może doprowadzić do niezgodnych lub niepoprawnych stanów.

Przykład

```
package Bounded_Queue_V2 is
-- Version 2, details of the queue type are hidden

   subtype Element_Type is Integer;

   type Queue_Type (Max_Size : Positive) is private;

   function Full (Queue : in Queue_Type) return Boolean;

   function Empty (Queue : in Queue_Type) return Boolean;

   function Size (Queue : in Queue_Type) return Natural;

   function First_Element (Queue : in Queue_Type) return Element_Type
      with
         Pre => not Empty (Queue);

   function Last_Element (Queue : in Queue_Type) return Element_Type
      with
         Pre => not Empty (Queue);

   procedure Clear (Queue : in out Queue_Type)
      with
         Post => Empty (Queue) and then Size (Queue) = 0;
```

Przykład cd.

```
  procedure Enqueue (Queue : in out Queue_Type;
                     Item  : in     Element_Type)
    with
      Pre  => not Full (Queue),
      Post => not Empty (Queue) and then
              Size (Queue) = Size (Queue'Old) + 1 and then
              Last_Element (Queue) = Item;

  procedure Dequeue (Queue : in out Queue_Type;
                     Item  :    out Element_Type)
    with
     Pre  => not Empty (Queue),
     Post => Item = First_Element (Queue'Old) and then
             Size (Queue) = Size (Queue'Old) - 1;

private

  type Queue_Array is array (Positive range <>) of Element_Type;
  type Queue_Type (Max_Size : Positive) is
    record
      Count : Natural  := 0;   -- Number of items
      Front : Positive := 1;   -- Index of first item
      Rear  : Positive := Max_Size;  -- Index of last item
      Items : Queue_Array (1 .. Max_Size);  -- The element array
    end record;

end Bounded_Queue_V2;
```

Pakiety rodzajowe

- W obu wersjach pakietu dla kolejek możliwe było przechowywanie w nich tylko liczb całkowitych.
- Aby przechowywać np. znaki trzeba by skopiować pliki i zamienić w nich typ `Integer` na `Character`.
- Język Ada umożliwia definiowanie ogólnych pakietów rodzajowych, dla których typy elementów są parametrami wymagającymi konkretyzacji przed użyciem.

Przykład

```
generic
   type Element_Type is private;
package Bounded_Queue is
-- Final version, generic with hidden details

   type Queue_Type (Max_Size : Positive) is private;

   function Full (Queue : in Queue_Type) return Boolean;

   function Empty (Queue : in Queue_Type) return Boolean;

   function Size (Queue : in Queue_Type) return Natural;

   function First_Element (Queue : in Queue_Type) return Element_Type
      with
         Pre => not Empty (Queue);

   function Last_Element (Queue : in Queue_Type) return Element_Type
      with
         Pre => not Empty (Queue);

   procedure Clear (Queue : in out Queue_Type)
      with
         Post => Empty (Queue) and then Size (Queue) = 0;
```

Przykład cd.

```
   procedure Enqueue (Queue : in out Queue_Type;
                      Item  : in     Element_Type)
      with
         Pre  => not Full (Queue),
         Post => not Empty (Queue) and then
                 Size (Queue) = Size (Queue'Old) + 1 and then
                 Last_Element (Queue) = Item;

   procedure Dequeue (Queue : in out Queue_Type;
                      Item  :    out Element_Type)
      with
        Pre  => not Empty (Queue),
        Post => Item = First_Element (Queue'Old) and then
                Size (Queue) = Size (Queue'Old) - 1;

private

   type Queue_Array is array (Positive range <>) of Element_Type;
   type Queue_Type (Max_Size : Positive) is
      record
         Count : Natural  := 0;    -- Number of items
         Front : Positive := 1;    -- Index of first item
         Rear  : Positive := Max_Size;  -- Index of last item
         Items : Queue_Array (1 .. Max_Size);  -- The element array
      end record;

end Bounded_Queue;
```

Przykład

```
with Bounded_Queue;
with Ada.Text_IO;    use Ada.Text_IO;
procedure Bounded_Queue_Example is
   -- Uses the generic version of the bounded queue package

   -- Instantiate a queue package with character elements
   package Char_Queue is new Bounded_Queue (Element_Type => Character);
   use Char_Queue;

   My_Queue : Char_Queue.Queue_Type (Max_Size => 100);
   Value    : Character;

begin
   Clear (My_Queue);  -- Initialize queue
   for Char in Character range 'f' .. 'p' loop
      Enqueue (Queue => My_Queue, Item => Char);
   end loop;
   for Count in Integer range 1 .. 5 loop
      Dequeue (Queue => My_Queue, Item => Value);
      Put (Value);
      New_Line;
   end loop;
   Clear (My_Queue);
   Put_Line ("Size of cleared queue is " & Integer'Image (Size (My_Queue)));
end Bounded_Queue_Example;
```

Pakiety definiujące zmienne

- Służą do manipulowania jednym obiektem.
- Deklarują wspólne dane dla innych pakietów.
- Nazywane są **singletonem**.

Przykład

pakiet definiujący

```
package Bingo_Numbers is

-- This package defines BINGO numbers and their associated letters

   -- The range of numbers on a Bingo Card
   type Bingo_Number is range 0 .. 75;

   -- 0 can't be called, it is only for the Free Play square
   subtype Callable_Number is Bingo_Number range 1 .. 75;

   -- Associations between Bingo numbers and letters
   subtype B_Range is Bingo_Number range  1 .. 15;
   subtype I_Range is Bingo_Number range 16 .. 30;
   subtype N_Range is Bingo_Number range 31 .. 45;
   subtype G_Range is Bingo_Number range 46 .. 60;
   subtype O_Range is Bingo_Number range 61 .. 75;

   -- The 5 Bingo letters
   type Bingo_Letter is (B, I, N, G, O);

end Bingo_Numbers;
```

Przykład

specyfikacja singletona

```
with Bingo_Numbers; use Bingo_Numbers;
package Bingo_Basket is

   function Empty return Boolean;

   procedure Load   -- Load all the Bingo numbers into the basket
      with
         Post => not Empty;

   procedure Draw (Letter : out Bingo_Letter;
                   Number : out Callable_Number)
   -- Draw a random number from the basket
      with
         Pre => not Empty;

end Bingo_Basket;
```

Przykład

implementacja singletona

```
with Ada.Numerics.Discrete_Random;
package body Bingo_Basket is

   type Number_Array is array (Callable_Number) of Callable_Number;
   The_Basket : Number_Array; -- A sequence of numbers in the basket
   The_Count  : Bingo_Number; -- The count of numbers in the basket

   package Random_Bingo is new Ada.Numerics.Discrete_Random
                                (Result_Subtype => Callable_Number);
   use Random_Bingo;
   -- The following object holds the state of a random Bingo number generator
   Bingo_Gen : Random_Bingo.Generator;

   procedure Swap (X : in out Callable_Number;
                   Y : in out Callable_Number) is
      Temp : Callable_Number;
   begin
      Temp := X;   X := Y;   Y := Temp;
   end Swap;

   function Empty return Boolean is (The_Count = 0);
   -- Example of an expression function
```

Przykład cd.

```
procedure Load is
  Random_Index : Callable_Number;
begin
  -- Put all numbers into the basket (in order)
  for Number in Callable_Number loop
    The_Basket (Number) := Number;
  end loop;
  -- Randomize the array of numbers
  Reset (Bingo_Gen);  -- Seed random generator from clock
  for Index in Callable_Number loop
    Random_Index := Random (Bingo_Gen);
    Swap (X => The_Basket (Index),
        Y => The_Basket (Random_Index));
  end loop;
  The_Count := Callable_Number'Last;  -- all numbers now in the basket
end Load;

procedure Draw (Letter : out Bingo_Letter;
            Number : out Callable_Number) is
begin
  Number := The_Basket (The_Count);
  The_Count := The_Count - 1;

  -- Determine the letter using the subtypes in Bingo_Definitions
  case Number is
    when B_Range =>  Letter := B;
    when I_Range =>  Letter := I;
    when N_Range =>  Letter := N;
    when G_Range =>  Letter := G;
    when O_Range =>  Letter := O;
  end case;
end Draw;
end Bingo_Basket;
```

Inicjowanie pakietu

- W przykładzie z grą w BINGO zanim przystąpi się do losowania należy załadować wszystkie liczby do koszyka.
- Ładowanie liczb powinno odbyć się tylko raz.

Przykład

```
   package Serial_Numbers is
      type Serial_Number is range 1000 .. Integer'Last;
      procedure Get_Next (Number : out Serial_Number);
   end Serial_Numbers;

package body Serial_Numbers is

   Next_Number : Serial_Number := Serial_Number'First;

   procedure Get_Next (Number : out Serial_Number) is
   begin
      Number := Next_Number;
      Next_Number := Next_Number + 1;
   end Get_Next;

end Serial_Numbers;
```

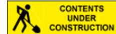
przetworzenie (ang. *elaboration*) polega na utworzeniu zdefiniowanych bytów

Przykład cd.

```
package Serial_Numbers2 is
   type Serial_Number is range 1000 .. Integer'Last;
   procedure Get_Next (Number : out Serial_Number);
end Serial_Numbers2;


package body Serial_Numbers2 is

   Next_Number : Serial_Number;

   procedure Get_Next (Number : out Serial_Number) is
   begin
      Number := Next_Number;
      Next_Number := Next_Number + 1;
   end Get_Next;

begin -- package initialization code
   Next_Number := Serial_Number'First;
end Serial_Numbers2;
```

Pakiety potomne

Przetworzenie



Trzy poziomy analizy w kolejności rosnącej ścisłości:

1. Pokazać, że program jest napisany w języku Ada z przestrzeganiem ograniczeń języka SPARK.
   - Prostym sposobem na zweryfikowanie tego jest kompilacja kodu kompilatorem zgodnym z **językiem SPARK** jak np. **GNAT**.
2. Pokazać, że program nie ma błędów zależności danych lub błędów przepływu informacji.
   - Można zweryfikować to używając **narzędzia SPARK**, które sprawdza (ang. **examine**) kod.
3. Pokazać, że program jest wolny od błędów wykonania gdyż spełnia wszystkie kontrakty, inwarianty i inne asercje.
   - Można to zweryfikować to używając **narzędzia SPARK**, które dowodzi (ang. **prove**) każdy plik ze źródłami.

Sprawdzanie zależności

- Kontrakty specyfikujące zależności między danymi i przepływy informacji.
- Służą weryfikacji następujących dwóch własności:
  1. żadne niezainicjowane dane nie są użyte,
  2. każdy wynik obliczeń programu faktycznie składa się w jakiś sposób na wynik końcowy.

Kontrakty zależności danych

- Opisują, od których globalnych danych zależy podprogram.
- Specyfikują czy podprogram czyta, zapisuje albo i czyta i zapisuje dane globalne.
- Do ich wyrażania stosuje się aspekt **Global**.

Przykład

```
pragma SPARK_Mode (On);
package Raster_Graphics is

   Workspace_Size : constant := 100;
   type Coordinate_Type is new Integer range 1 .. Workspace_Size;
   type Point is
      record
         X, Y : Coordinate_Type;
      end record;

   type Line_Algorithm_Type is (Bresenham, Xiaolin_Wu);
   type Status_Type is (Success, Line_Too_Short, Algorithm_Not_Implemented);

   Status         : Status_Type;
   Line_Algorithm : Line_Algorithm_Type;
   Line_Count     : Natural;

   procedure Draw_Line (A, B : in Point)
     with
      Global => (Input  => Line_Algorithm,
                 Output => Status,
                 In_Out => Line_Count);

end Raster_Graphics;
```

Przykład

```
pragma SPARK_Mode(On);
package body Raster_Graphics is

   procedure Draw_Line(A, B : in Point) is

      Min_Distance : constant Coordinate_Type := 2;

      -- Verify that A, B are far enough apart.
      -- Write error code to Status if not.
      procedure Check_Distance
        with
         Global => (Input  => (A, B),
                    Output => Status)
      is
         Delta_X : Coordinate_Type := abs (A.X - B.X);
         Delta_Y : Coordinate_Type := abs (A.Y - B.Y);
      begin
         if Delta_X**2 + Delta_Y**2 < Min_Distance**2 then
            Status := Line_Too_Short;
         else
            Status := Success;
         end if;
      end Check_Distance;
```

Przykład cd.

```
   begin
      Check_Distance;
      if Status = Success then
         case Line_Algorithm is
            when Bresenham =>
               -- Algorithm implementation not shown...
               Line_Count := Line_Count + 1;

            when Xiaolin_Wu =>
               Status := Algorithm_Not_Implemented;
         end case;
      end if;
   end Draw_Line;

end Raster_Graphics;
```

Kontrakty przepływu informacji

- Ważną częścią analizy przepływu jest śledzenie które wartości są użyte do obliczenia których rezultatów.
  - Aho, Alfred V., Lam, Monica S., Sethi, Ravi, and Ullman, Jeffrey D. 2007. **Compilers Principles, Techniques & Tools**. 2nd edn. Boston, MA, Addison Wesley.

Depends => (OUTPUT_PART => INPUT_PART)

Przykład

```
procedure Search (Text     : in  String;
                  Letter   : in  Character;
                  Start    : in  Position;
                  Found    : out Boolean;
                  Position : out Position)
  with
    Global => null,
    Depends => (Found    => (Text, Letter, Start),
                Position => (Text, Letter, Start));
```

Przykład

```
procedure Initialize (Value : out Integer)
  with
    Depends => (Value => null);
```

← nie zależy od niczego

Przykład

```
procedure Update (Value  : in out Integer;
                  Adjust : in     Integer)
  with
    Depends => (Value =>+ null,
                null  => Adjust);
```

zależy od siebie i niczego innego (→ Value =>+ null)

nic nie zależy od Adjust (→ null => Adjust)

Żaden wynik nie zależy od **Adjust** ale dzięki aspektowi **Depends** możliwe jest wyspecyfikowanie, że może będzie kiedyś dopisany odpowiedni fragment kodu używający tej wartości.

Skróty w aspekcie **Depends**.

Przykład

```
with
  Global => null,
  Depends => (Found    => (Text, Letter, Start),
              Position => (Text, Letter, Start));

with
  Global => null,
  Depends => ((Found, Position) => (Text, Letter, Start));
```

Przykład

```
with
  Depends => (Value => (Text, Name, Value))

with
  Depends => (Value =>+ (Text, Name))
```

Zarządzanie stanem

Przykład

Program przykładowy dostarczany z narzędziem **SPARK**.

```
package Segway
  with SPARK_Mode,
       Abstract_State => State
is

   type Status_Type is (Still, Forward, Backward);

   type Input is (No_Input, Nothing, Accelerate, Brake);

   subtype Valid_Input is Input range Nothing .. Brake;

   Max_Speed : constant := 100;

   type Speed_Type is range -Max_Speed .. Max_Speed;

   function Current_Status return Status_Type
     with Global => State;

   function Current_Speed return Speed_Type
     with Global => State;
```

Przykład cd.

```
   function Speed_Is_Valid return Boolean
   --  Expresses the required invariant relationship between
   --  Current_Speed and Current_Status
   is
      (case Current_Status is
         when Still    => Current_Speed = 0,
         when Forward  => Current_Speed > 0,
         when Backward => Current_Speed < 0)
     with Global => State;

   procedure Initialize
     with Global  => (Output => State),
          Depends => (State => null),
          Post    => Speed_Is_Valid and Current_Status = Still;
   --  Initializes State and establishes initial invariant

   procedure State_Update (I : Valid_Input)
     with Global  => (In_Out => State),
          Depends => (State => (State, I)),
          Pre     => Speed_Is_Valid,
          Post    => Speed_Is_Valid;

   procedure Halt
     with Global  => (In_Out => State),
          Depends => (State => State),
          Pre     => Speed_Is_Valid,
          Post    => Speed_Is_Valid and Current_Status = Still;

end Segway;
```

Przykład cd.

```
package body Segway
  with SPARK_Mode,
       Refined_State => (State => (Speed, Status))
is
   ----------------------------------------------------
   --          SPARK 2014 - Segway Example           --
   --                                                --
   -- This example illustrates the use of Ada2012    --
   -- expression functions to specify an invariant   --
   -- that must be maintained in a state-machine     --
   -- package                                        --
   ----------------------------------------------------

   --  Actual concrete states are declared here in the body
   --  to prevent direct tampering by clients
   Speed  : Speed_Type;
   Status : Status_Type;

   --------------------
   -- Current_Status --
   --------------------

   function Current_Status return Status_Type is (Status)
     with Refined_Global => Status;
```

Przykład cd.

```
   -------------------
   -- Current_Speed --
   -------------------

   function Current_Speed return Speed_Type is (Speed)
     with Refined_Global => Speed;

   ----------------
   -- Initialize --
   ----------------

   procedure Initialize
     with Refined_Global => (Output => (Status, Speed)),
          Refined_Depends => ((Status, Speed) => null),
          Refined_Post    => Status = Still and Speed = 0
   is
   begin
      Status := Still;
      Speed  := 0;
   end Initialize;
```

Przykład cd.

```
------------------
-- State_Update --
------------------

procedure State_Update (I : Valid_Input)
  with Refined_Global => (In_Out => (Status, Speed)),
       Refined_Depends => (Speed => (Speed, I),
                           Status => (Status, Speed, I))
is
begin
   case I is
      when Nothing =>
         null;
      when Accelerate =>
         if Speed < Speed_Type'Last then
            Speed := Speed + 1;
         end if;
      when Brake =>
         if Speed > Speed_Type'First then
            Speed := Speed - 1;
         end if;
   end case;
   if Speed = 0 then
      Status := Still;
   elsif Speed = 1 and then I = Accelerate then
      Status := Forward;
   elsif Speed = -1 and then I = Brake then
      Status := Backward;
   end if;
end State_Update;
```

Domyślne inicjowanie

Przykład cd.

```
----------
-- Halt --
----------

procedure Halt
  with Refined_Global => (In_Out => (Status, Speed)),
       Refined_Depends => (Speed => (Speed, Status),
                           Status => (Speed, Status))
is
begin
   while Status /= Still loop
      pragma Loop_Invariant (Speed_Is_Valid);
      if Speed > 0 then
         State_Update (Brake);
      else
         State_Update (Accelerate);
      end if;
   end loop;
end Halt;

end Segway;
```

Synteza kontraktów zależności

Dowodzenie

- Dowody tworzone przez narzędzia **SPARK** są całkowicie statyczne.
- Jeśli coś zostanie udowodnione z powodzeniem, to będzie prawdziwe dla <u>każdego możliwego</u> obliczenia bez względu na dostarczone programowi dane.
- To krytyczna własność dowodzenia odróżniająca go od testowania.
- Jeśli jakieś asercje nie mogły być udowodnione, to można je sprawdzać w trakcie testowania tak aby zapewnić co do nich pewien poziom zaufania.

Predefiniowane wyjątki

- Język **SPARK** pozwala programiście wprost zgłosić wyjątek instrukcją **raise** ale narzędzia **SPARK** starają się formalnie udowodnić, że nigdy nie zostanie ona wykonana.
- Jedyne wyjątki jakie mogą wystąpić w **SPARK**, to omówione poniżej cztery predefiniowane wyjątki generowane automatycznie przez kompilator gdy program nie spełnia zasad języka.

Program_Error

- Wyjątek Program_Error jest zgłaszany w języku Ada w każdej sytuacji stwarzającej problem, która nie mogła być wykryta przez kompilator.

Przykład

```
subtype Index_Type is Natural range 0 .. 1023;
type Integer_Array is array (Index_Type) of Integer;

function Search_For_Zero (Values : in Integer_Array ) return Index_Type is
begin
  for Index in Index_Type loop
    if Value (Index) = 0 then
      return Index;
    end if;
  end loop;
end Search_For_Zero;
```

- Jeśli funkcja osiągnie końcowy **end**, to zgłaszany jest wyjątek **Program_Error** (ma to miejsce w sytuacji gdy w tablicy nie ma ani jednego zera).
- W niebezpiecznych językach, jak **C**, błędy wykonania powodują nieprzewidywalne zachowanie programu.
- W językach bezpiecznych, jak **Ada** czy jej podzbiór **SPARK**, każdy błąd wykonania kończy się wyjątkiem.
- W systemach krytycznych nie jest dopuszczalne wystąpienie błędu wykonania.
- W **SPARK** dowodzi się formalnie, że program jest wolny od błędów wykonania, zatem zbędna jest w nim obsługa wyjątków.

Przykład cd.

- Poprzednia wersja funkcji **Search_For_Zero** nie jest dopuszczalna w języku **SPARK** gdyż dopuszcza zgłoszenie niejawnie dodany wyjątek przed końcowym **end**.
- Aby temu zaradzić należy albo dopisać instrukcję **return** zwracającą wartość wyrażającą brak zera albo dopisać warunek wstępny **Pre** i jawne zgłoszenie wyjątku.

```
subtype Index_Type is Natural range 0 .. 1023;
type Integer_Array is array (Index_Type) of Integer;

function Search_For_Zero (Values : in Integer_Array ) return Index_Type
with Pre => (for some Index in Index_Type => Values (Index) = 0)
is
begin
  for Index in Index_Type loop
    if Value (Index) = 0 then
      return Index;
    end if;
  end loop;
  raise Program_Error;
end Search_For_Zero;
```

SPARK formalnie udowodni, że ten wyjątek nigdy nie będzie zgłoszony

Tasking_Error

- Język **Ada** wspiera pisanie programów zbudowanych z wielu współbieżnych zadań (ang. *task*).
- Pewne błędne sytuacje mogą zajść podczas wykonywania takich zadań co sygnalizowane jest wyjątkiem **Tasking_Error**.
- Obecna wersja **SPARK 2014** nie dopuszcza współbieżnych zadań, zatem wyjątek **Tasking_Error** nie zostanie w niej nigdy zgłoszony.
- **SPARK** umożliwia mieszanie fragmentów w języku **Ada** i jego podzbiorze **SPARK**, przez co możliwa jest formalna analiza sekwencyjnych fragmentów większego współbieżnego programu.

Storage_Error

- W programach w języku **Ada** wyjątek **Storage_Error** jest zgłaszany gdy wyczerpie się pamięć.
- Możliwe jest to w dwóch przypadkach:
  - Wyczerpała się sterta (ang. *heap*) dla obiektów alokowanych dynamicznie.
  - Wyczerpał się stos (ang. *stack*) dla lokalnych zmiennych podczas wywołania podprogramu.
- Pierwszy przypadek w języku **SPARK** nie jest możliwy, bo nie dopuszcza korzystania z alokacji pamięci na stercie.
- Drugi przypadek jest trudniejszy i wymaga dwóch kroków:
  - Wyeliminowanie wszelkiej rekurencji (bezpośredniej i pośredniej). Narzędzia **SPARK** nie wykrywają rekurencji ale dodatkowe narzędzia jak **GNATcheck** od **AdaCore** to robi.
  - Gdy nie ma rekurencji, buduje się **drzewo wywołań** i dla każdej ścieżki obliczeń oszacowuje się wielkość użytego stosu. Narzędzia **SPARK** tego nie robią ale **GNATstack** od **AdaCore** i **StackAnalyzer** od **AbsInt** to potrafią.

Constraint_Error

- Wyjątek **Constraint_Error** jest zgłaszany gdy:
  - wartość przypisywana zmiennej wykracza poza zakres typu zmiennej,
  - odwołano się do elementu tablicy wartością indeksu wykraczającą poza zdefiniowany zakres indeksu,
  - wystąpiło przepełnienie (ang. *overflow*) podczas operacji arytmetycznych.
- Pewne powodu **Constraint_Error** w języku **Ada** nie mogą pojawić się w języku **SPARK** z powodu jego ograniczeń (np. odwołanie się danej wskazywanej pustym wskaźnikiem).
- Wiele innych sytuacji jest możliwych w języku **SPARK** i wymaga formalnych dowodów, że nie zaistnieją (np. przekroczenie zakresu wartości albo zakresu indeksu).

- Dla każdego punktu w programie gdzie wymagana jest kontrola, narzędzia **SPARK** generują **warunek weryfikacji** (ang. *verification condition*).
- Każdy warunek weryfikacji jest w postaci implikacji.
- Przesłanki (hipotezy) warunku weryfikacji są wzięte z kodu prowadzącego do danego punktu programu z uwzględnieniem wszystkich możliwych ścieżek obliczeń.
- Konkluzją warunku weryfikacji jest że wyjątek nie zostanie zgłoszony, tj. wyraża, że warunek, który mógłby spowodować wyjątek jest fałszywy.
- W typowym programie generowana jest duża liczba warunków weryfikacji o bardzo złożonych przesłankach.
- Do weryfikacji warunków wykorzystywane są programy automatycznie dowodzące twierdzenia.
- Często wymagają one jednak pomocy ze strony człowieka np. przez przeformułowanie programu albo dostarczenie dodatkowych informacji (asercji, inwariantów itp.).

Kontrakty

Aspekt/Pragma	w SPARK?	tylko w SPARK?
Assert	**tak**	nie
Assert_Or_Cut	**tak**	**tak**
Assume	**tak**	**tak**
Contract_Cases	**tak**	**tak**
Dynamic_Predicates	nie	nie
Initial_Condition	**tak**	**tak**
Loop_Invariant	**tak**	**tak**
Loop_Variant	**tak**	**tak**
Post	**tak**	nie
Pre	**tak**	nie
Refined_Post	**tak**	**tak**
Static_Predicate	**tak**	nie
Type_Invariant	nie	nie

Przykład

```
function Fibonacci (N : in Natural) return Natural;


function Fibonacci (N : in Natural) return Natural
  with
    Pre => N <= 46;


subtype Fibonacci_Argument_Type us Natural range 0 .. 46;

function Fibonacci (N : in Fibonacci_Argument_Type) return Natural;
```

Aspekt **Pre**

Warunek wstępny (ang. *precondition*) musi być spełniony w chwili wywołania podprogramu.

Przykład

```
pragma SPARK_Mode(On);
package Shapes is

   subtype Coordinate_Type is Integer range -100 .. +100;
   subtype Radius_Type     is Coordinate_Type range 0 .. 10;

   type Circle is
      record
         Center_X : Coordinate_Type;
         Center_Y : Coordinate_Type;
         Radius   : Radius_Type;
      end record;

   -- Return True if X, Y are inside circle C.
   function Inside_Circle (X, Y : in Coordinate_Type;
                           C    : in Circle)  return Boolean
     with
       Pre => C.Center_X + C.Radius in Coordinate_Type and
              C.Center_X - C.Radius in Coordinate_Type and
              C.Center_Y + C.Radius in Coordinate_Type and
              C.Center_Y - C.Radius in Coordinate_Type;
end Shapes;
```

Przykład

```
package Searchers
  with SPARK_Mode => On
is
   subtype Index_Type is Positive range 1 .. 100;
   type Array_Type is array(Index_Type) of Integer;

   procedure Binary_Search (Search_Item : in  Integer;
                            Items       : in  Array_Type;
                            Found       : out Boolean;
                            Result      : out Index_Type)
      with
         Pre =>
            (for all J in Items'Range =>
               (for all K in J + 1 .. Items'Last => Items(J) <= Items(K)));
end Searchers;
```

```
Pre => (for all J in Items'First .. Items'Last - 1 =>  Items (J) <= Items (J+1))
```

Aspekt **Post**

Warunek końcowy (ang. *postcondition*) musi być spełniony w chwili zakończenia działania podprogramu.

Przykład

```
package Searchers2
  with SPARK_Mode => On
is
   subtype Index_Type is Positive range 1 .. 100;
   type Array_Type is array(Index_Type) of Integer;

   procedure Binary_Search (Search_Item : in  Integer;
                            Items       : in  Array_Type;
                            Found       : out Boolean;
                            Result      : out Index_Type)

     with
       Pre =>
         (for all J in Items'Range =>
            (for all K in J + 1 .. Items'Last => Items(J) <= Items(K))),
       Post =>
         (if Found then Search_Item = Items (Result)
                   else (for all J in Items'Range => Search_Item /= Items(J)));

end Searchers2;
```

Dualizm warunków wstępnych i końcowych

- Wywoływacz jest zainteresowany jak najsłabszym warunkiem wstępnym, gdyż wtedy najłatwiej go udowodnić.
- Z drugiej strony, wywoływacz jest zainteresowany jak najsilniejszym warunkiem końcowym, gdyż dostarcza on najwięcej informacji możliwej do użyciu po powrocie z podprogramu.
- Programista jest zainteresowany w jak najsilniejszym warunku wstępnym, gdyż dostarcza on najwięcej informacji do podprogramu.
- Z drugiej strony, programista jest zainteresowany w jak najsłabszym warunku końcowym, bo najłatwiej go udowodnić.

Konieczne jest zachowanie równowagi

- **Stan wstępny podprogramu** (ang. *prestate*), to stan całego programu na chwilę przed rozpoczęciem wykonywania się podprogramu.
- **Stan końcowy podprogramu** (ang. *poststate*), to stan programu po powrocie z podprogramu.

Przykład

```
procedure Smallest_Factor (N      : in out Positive;
                           Factor : out Positive)
  with
    Post => Is_Prime(Factor) and
            (N = N'Old / Factor) and
            (N'Old rem Factor = 0);
```

	Opis
A’Old(Index)	Odwołuje się do oryginalnego (prestate) elementu na pozycji podanej przez aktualny indeks (poststate).
A(Index’Old)	Odwołuje się do elementu aktualnej (poststate) tablicy na pozycji oryginalnego (prestate) indeksu.
A’Old(Index’Old)	Odwołuje się do elementu oryginalnej (prestate) tablicy na oryginalnej (prestate) pozycji indeksu. Cała oryginalna tablica i oryginalny indeks jest zachowany gdy następuje wejście do podprogramu.
A(Index)’Old	To samo co A’Old(Index’Old) ale tylko oryginalna wartość A(Index) jest zachowywana (nie cała tablica).

Przykład

```
procedure Smallest_Factor (N : in out Positive;
                           Factor : out Positive)
  with
    Post => (N = N'Old / Factor) and
            (N'Old rem Factor = 0) and
            (for all J in 2 .. Factor - 1 => N'Old rem J /= 0);
```

```
function Smallest_Factor (N : in Positive) return Positive
  with
    Post => (N rem Smallest_Factor'Result = 0) and
            (for all J in 2 .. Smallest_Factor'Result - 1 => N rem J /= 0);
```

Warunki początkowe pakietu

Prywatne informacje

Jeśli ukryto szczegóły implementacji w części private, to trzeba sprawdzanie warunków ukryć w implementacji funkcji publicznej.

Przykład

w kontrakcie nie można użyć danych zdefiniowanych w części **private**

```
pragma SPARK_Mode(On);
package Shapes2 is
   subtype Coordinate_Type is Integer range -100 .. +100;
   subtype Radius_Type     is Coordinate_Type range 0 .. 10;
   type Circle is private;

   -- Return True if X, Y are inside circle C.
   function Inside_Circle (X, Y : in Coordinate_Type;
                           C    : in Circle) return Boolean
      with Pre => In_Bounds (C);

   -- Return True if C is entirely in the workspace.
   function In_Bounds (C : in Circle) return Boolean;

private
   type Circle is
      record
         Center_X : Coordinate_Type;
         Center_Y : Coordinate_Type;
         Radius   : Radius_Type;
      end record;
end Shapes2;
```

Niezmienniki typów

Na razie SPARK nie obsługuje niezmienników typów.

Przykład

```
type Circle is private
   with
      Type_Invariant => In_Bounds (Circle);
```

Predykaty statyczne

Umożliwiają np. definiowanie podtypów z nieciągłymi zakresami wartości.

Przykład

```
pragma SPARK_Mode(On);
package Scrabble is

   subtype Scrabble_Letter is Character range 'A' .. 'Z';

   subtype Scrabble_Value is Positive
      with Static_Predicate => Scrabble_Value in 1 .. 5 | 8 | 10;

   type Scrabble_Word is array(Positive range <>) of Scrabble_Letter;

   subtype Scrabble_Score is Natural range 0 .. 100;
   function Raw_Score (Word : in Scrabble_Word) return Scrabble_Score
      with Pre => (Word'Length <= 10);

end Scrabble;
```

Predykaty dynamiczne

Na razie SPARK nie obsługuje dynamicznych predykatów.

Przykład

```
subtype Even_Type is Natural
  with Dynamic_Predicate => Even_Type mod 2 = 0;

subtype Prime_Type is Natural
  with Dynamic_Predicate -> Is_Prime (Prime_Type);

type Lower_Half is
    record
      X : Natural;
      Y : Natural;
    end record
  with Dynamic_Predicate => Lower_Half.X > Lower_Half.Y;
```

Przypadki kontraktów

Używając przypadków kontraktów można podzielić dziedzinę wartości parametrów wejściowych na rozłączne podzbiory i z każdym z nich związać odpowiednie warunki końcowe.

Przykład

```
subtype Sign_Type is Integer range -1 .. 1;

function Sign (X : in Integer) return Sign_Type
  with
    Contract_Cases =>
      (X < 0 => Sign'Result = -1,
       X = 0 => Sign'Result =  0,
       X > 0 => Sign'Result =  1);
```

Pragma **Assert**

Pozwala wyrazić warunki jakie uważamy, że w danym miejscu są prawdziwe. Pragma może wystąpić w dowolnym miejscu podprogramu gdzie są deklaracje lub instrukcje. Każda asercja wymaga udowodnienia, że zawsze jest prawdziwa w danym miejscu.

Przykład

```
pragma Assert (X > 1);
```

Przykład cd.

```
-- Compute health insurance premium.
if not Student.Self_Insured then
   Insurance := 1_000.00;
end if;

pragma Assert_And_Cut ((Tuition   in 0.00 .. 30_000.00) and
                       (Insurance in 0.00 ..  1_000.00));

-- Compute base fees depending on full-time/part-time status.
if Student.Part_Time then
   Fees := 100.00;
else
   Fees := 500.00;
end if;

-- Room and board.
if Student.Resident then
   Fees := Fees + 4_000.00;  -- Room.
   case Student.Meal_Plan is
      when None      => null;
      when On_Demand => Fees := Fees +   100.00;
      when Basic     => Fees := Fees + 1_000.00;
      when Full      => Fees := Fees + 3_000.00;
   end case;
else
   -- Non resident students getting a meal plan pay a premium.
   case Student.Meal_Plan is
      when None      => null;
      when On_Demand => Fees := Fees +   200.00;
      when Basic     => Fees := Fees + 1_500.00;
      when Full      => Fees := Fees + 4_500.00;
   end case;
end if;
```

Pragma **Assert_And_Cut**

Odpowiada pragmie Assert z tą różnicą, że jeśli do miejsca wystąpienia pragmy Assert_And_Cut dochodziło wiele ścieżek obliczeń, to wychodzi z niego tylko jedna ścieżka.

Przykład

```
package body Students3
   with SPARK_Mode => On
is
   function Compute_Bill (Student      : in Student_Record;
                          Base_Tuition : in Money_Type) return Money_Type is
      Tuition   : Money_Type;
      Fees      : Money_Type;
      Grants    : Money_Type := 0.00;
      Insurance : Money_Type := 0.00;
   begin
      Tuition := Base_Tuition;

      if not Student.In_State then
         -- Out of state tuition is 50% higher.
         Tuition := Tuition + Tuition / 2;
      end if;

      pragma Assert_And_Cut (Tuition in 0.00 .. 30_000.00);
```

Przykład cd.

```
      pragma Assert_And_Cut ((Tuition   in   0.00 .. 30_000.00) and
                             (Insurance in   0.00 ..  1_000.00) and
                             (Fees      in 100.00 ..  7_500.00));

      -- University policy: give high achieving students a break.
      if Student.GPA >= 3.00 then
         Grants := Grants + 250.00;

         -- Special directive from the state for very high achieving women.
         if Student.GPA >= 3.75 and Student.Gender = Female then
            Grants := Grants + 250.00;
         end if;
      end if;

      pragma Assert_And_Cut ((Tuition   in   0.00 .. 30_000.00) and
                             (Insurance in   0.00 ..  1_000.00) and
                             (Fees      in 100.00 ..  7_500.00) and
                             (Grants    in   0.00 ..    500.00));

      return ((Tuition + Fees) - Grants) + Insurance;
   end Compute_Bill;

end Students3;
```

Pragma **Assume**

Podobnie jak pragma **Assert** wyraża warunek jaki uważamy, że jest prawdziwy w danym miejscu ale nie wymaga udowodnienia. SPARK będzie używał tego warunku jako założenie do dowodzenia innych warunków weryfikacji.

Przykład

```
pragma Assume (C > 0);
A := B / C;
```

```
if Clock_Value = Clock_Type'Last then
  Restart_System;
else
  Clock_Value := Clock_Value + 1;
end if;
```

```
pragma Assume (Clock_Value < Clock_Type'Last);
Clock_Value := Clock_Value + 1;
```

Przykład cd.

```
package body Searchers2
  with SPARK_Mode => On
is

   procedure Binary_Search (Search_Item : in  Integer;
                            Items       : in  Array_Type;
                            Found       : out Boolean;
                            Result      : out Index_Type) is

      Low_Index  : Index_Type := Items'First;
      Mid_Index  : Index_Type;
      High_Index : Index_Type := Items'Last;
   begin
      Found  := False;
      Result := Items'First;  -- Initialize Result to "not found" case.

      -- If the item is out of range, it is not found.
      if Search_Item < Items(Low_Index) or Items(High_Index) < Search_Item then
         return;
      end if;
```

Niezmienniki pętli

Pętla dostarcza potencjalnie nieznaną liczbę ścieżek obliczeń. Dzięki niezmiennikom pętli możliwe jest wyrażenie w pętli warunku, który jest prawdziwy przy pierwszym i każdym kolejnym osiągnięciem miejsca, w którym go podano.

Przykład

```
package Searchers2
  with SPARK_Mode => On
is
   subtype Index_Type is Positive range 1 .. 100;
   type Array_Type is array(Index_Type) of Integer;

   procedure Binary_Search (Search_Item : in  Integer;
                            Items       : in  Array_Type;
                            Found       : out Boolean;
                            Result      : out Index_Type)
     with
       Pre =>
         (for all J in Items'Range =>
            (for all K in J + 1 .. Items'Last => Items(J) <= Items(K))),
       Post =>
         (if Found then Search_Item = Items (Result)
                   else (for all J in Items'Range => Search_Item /= Items(J)));

end Searchers2;
```

Przykład cd.

```
      loop
         Mid_Index := (Low_Index + High_Index) / 2;
         if Search_Item = Items(Mid_Index) then
            Found  := True;
            Result := Mid_Index;
            return;
         end if;

         exit when Low_Index = High_Index;

         pragma Loop_Invariant
           (Search_Item in Items(Low_Index) .. Items(High_Index));

         if Items(Mid_Index) < Search_Item then
            Low_Index := Mid_Index;
         else
            High_Index := Mid_Index;
         end if;

      end loop;
   end Binary_Search;

end Searchers2;
```

gdy **High_Index = Low_Index+1** i **Items(Mid_Index) < Search_Item**, wówczas **Low_Index** pozostaje niezmieniony

Zmienniki pętli

Umożliwiają wyrażenie warunków, z których wynika, że wykonywanie pętli kończy się.

Przykład

```
package Searchers3
  with SPARK_Mode => On
is
   subtype Index_Type is Positive range 1 .. 100;
   type Array_Type is array(Index_Type) of Integer;

   procedure Binary_Search (Search_Item : in  Integer;
                            Items       : in  Array_Type;
                            Found       : out Boolean;
                            Result      : out Index_Type)
      with
         Pre =>
            (for all J in Items'Range =>
               (for all K in J + 1 .. Items'Last => Items(J) <= Items(K))),
         Post =>
           (if Found then Search_Item = Items(Result)
                     else (for all J in Items'Range => Search_Item /= Items(J)));

end Searchers3;
```

Przykład cd.

```
package body Searchers3
  with SPARK_Mode => On
is
   procedure Binary_Search (Search_Item : in  Integer;
                            Items       : in  Array_Type;
                            Found       : out Boolean;
                            Result      : out Index_Type) is
      Low_Index  : Index_Type := Items'First;
      Mid_Index  : Index_Type;
      High_Index : Index_Type := Items'Last;
   begin
      Found  := False;
      Result := Items'First;  -- Initialize Result to "not found" case.

      -- If the item is out of range, it is not found.
      if Search_Item < Items(Low_Index) or Items(High_Index) < Search_Item then
         return;
      end if;
```

Przykład cd.

```
            loop
               Mid_Index := (Low_Index + High_Index) / 2;
               if Search_Item = Items(Mid_Index) then
                  Found  := True;
                  Result := Mid_Index;
                  return;
               end if;

               exit when Low_Index = High_Index;

               pragma Loop_Invariant (not Found);
               pragma Loop_Invariant (Mid_Index in Low_Index .. High_Index - 1);
               pragma Loop_Invariant (Items(Low_Index) <= Search_Item);
               pragma Loop_Invariant (Search_Item <= Items(High_Index));
               pragma Loop_Variant (Decreases => High_Index - Low_Index);

               if Items(Mid_Index) < Search_Item then
                  if Search_Item < Items(Mid_Index + 1) then
                     return;
                  end if;
                  Low_Index := Mid_Index + 1;
               else
                  High_Index := Mid_Index;
               end if;

            end loop;
         end Binary_Search;

      end Searchers3;
```