EBook Gratuito

APPRENDIMENTO ada

Free unaffiliated eBook created from
Stack Overflow contributors.

#ada

Sommario

Di

Capitolo 1: Iniziare con Ada

Osservazioni

Ada è un linguaggio di programmazione per computer orientato agli oggetti di livello internazionale, standardizzato ad alto livello che supporta una forte digitazione e una programmazione strutturata. Ulteriori informazioni possono essere trovate qui .

Versioni

Versione	Data di rilascio
Ada 2012 (TC-1)	2016/04/01
Ada 2012	2012/12/10
Ada 2005	2007-01-01
Ada 95	1995/12/10
Ada 83	1983/01/01

Examples

Installazione o configurazione

Ada è un linguaggio di programmazione per il quale esistono più compilatori.

- Uno di questi compilatori, e forse il più usato, è GNAT. Fa parte della toolchain GCC. Può essere installato da diverse fonti:
    - La release GPL annuale fatta da AdaCore, disponibile gratuitamente sul sito libre . Questa versione ha subito tutti i test interni che AdaCore fa per le sue versioni professionali, è disponibile su un gran numero di piattaforme. Il compilatore e il suo runtime sono rilasciati sotto la licenza GPL e, a meno che non si utilizzi nessun runtime, tutti gli eseguibili che si distribuiscono saranno coperti da questa licenza. Per gli accademici e i progetti nelle loro fasi iniziali, questo non è un problema.
    - Gcc di FSF riceve regolarmente le stesse patch. La versione di GNAT potrebbe non essere sempre aggiornata, ma si raggiunge regolarmente.
    - Un certo numero di contributori sta confezionando la versione di FSF per varie distribuzioni Linux (sistemi basati su Debian, tra gli altri) e binari per Mac OS X. L'uso del gestore di pacchetti dalla distribuzione potrebbe essere il modo più semplice per installare GNAT. Tali versioni sono fornite con la licenza GCC standard e consentono

di scrivere codice sorgente chiuso.

- AdaCore fornisce anche GNAT Pro , che viene fornito con la licenza GCC standard che consente di scrivere codice sorgente chiuso. Ancora più importante, forse, viene fornito con il supporto, in caso di domande sull'uso del linguaggio, sugli strumenti, su come implementare al meglio qualcosa e, naturalmente, segnalazioni di bug e richieste di miglioramento.

Un altro numero di compilatori è elencato nell'Ada WikiBook , insieme alle istruzioni di installazione. Getadanow.com presenta edizioni di GNAT FSF, pronte per vari sistemi operativi su diversi tipi di hardware o macchine virtuali. Il sito raccoglie anche risorse per l'apprendimento e la condivisione di Ada.

Ciao mondo

```
with Ada.Text_IO;

procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("Hello World");
end Hello_World;
```

In alternativa, dopo aver importato il pacchetto Ada.Text_IO , puoi `use Ada.Text_IO;` per poter utilizzare Put_Line senza dichiarare esplicitamente il pacchetto da cui proviene, in quanto tale:

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

procedure Hello_World is
begin
    Put_Line ("Hello World");
end Hello_World;
```

Se stai usando il compilatore `gnat` , questo semplice programma può essere compilato con

```
gnatmake hello_world
```

Questo genererà un numero di file, incluso un `hello_world` (o `hello_world.exe` su Windows) che puoi eseguire per vedere il famoso messaggio. Il nome dell'eseguibile viene calcolato automaticamente dal nome del sottoprogramma principale Ada. In Ada un sottoprogramma principale può avere qualsiasi nome. Deve solo essere una procedura senza parametri, che si fornisce come argomento per `gnatmake` .

Altri compilatori hanno requisiti simili, anche se ovviamente il comando di compilazione è diverso.

Versione

Il linguaggio di programmazione Ada standard è definito nel *Manuale di riferimento Ada* . Le modifiche alla versione provvisoria e le note di rilascio sono trattate nei documenti razionali corrispondenti. Le implementazioni tipicamente documentano la loro conformità allo standard sotto

forma di guida per l'utente e / o manuale di riferimento, ad esempio .

- Ada 2012
    - *Manuale di riferimento della lingua Ada 2012*
    - *Motivazione per Ada 2012*
- Ada 2005
    - *Manuale di riferimento della lingua Ada 2005*
    - *Motivazione per Ada 2005*
- Ada 95
    - *Ada 95 Language Reference Manua* l
    - *Motivazione per Ada 95*
- Ada 83
    - *Ada 83 Guida di riferimento Manua* l
    - *Ada 83 Razionale per la progettazione del linguaggio di programmazione Ada®*

biblioteche

Come per qualsiasi linguaggio di programmazione, Ada viene fornito con ampie librerie per svolgere varie attività. Ecco alcuni suggerimenti su alcuni di essi, anche se la ricerca su Github ne porterà altri.

- Lo stesso runtime di Ada, distribuito da tutti i compilatori, include un ampio set di pacchetti e allegati, che vanno da strutture e contenitori di dati, input / output, manipolazione di stringhe, manipolazione del tempo, file, calcoli numerici, multi-tasking, commutatori a linea di comando, numeri casuali, ...
- Il compilatore GNAT viene fornito con un proprio runtime esteso, con nuovi pacchetti nella gerarchia `GNAT` , che forniscono supporto per espressioni regolari, ordinamento, ricerca, unicode, CRC, input / output tempo, ...
- gnatcoll è una libreria disponibile dal sito libre di AdaCore e include un ampio framework di registrazione, estendendo le applicazioni con python, mmap, un ampio framework per interfacciarsi con i file system, analizzare messaggi e-mail e cassette postali, un ampio framework per interagire con i database in un tipo sicuro, interfaccia a varie librerie come icona, readline, colori terminali, supporto per tipi conteggiati di riferimento per la gestione automatica della memoria, file JSON, ...
- XML / Ada è una libreria per analizzare e convalidare documenti XML
- GtkAda è un binding completo alla libreria gtk +, che consente di scrivere interfacce utente portatili su Unix, Windows e OSX.
- AWS è un framework per creare server Web in Ada, con supporto completo per vari

protocolli come HTTP, Websockets, ... e il proprio sistema di template.

Leggi Iniziare con Ada online: https://riptutorial.com/it/ada/topic/3900/iniziare-con-ada

Capitolo 2: Attuazione del modello produttore-consumatore

introduzione

Una dimostrazione di come il modello produttore-consumatore è implementato in Ada.

Sintassi

- function Scalar ' Image (Argument: Scalar'Base) return String;
- compito Task_Name;
- task Task_Name è Entries end;
- task body Task_Name is Dichiarazioni iniziano Code end;
- voce Entry_Name;
- accetta Entry_Name;
- Uscita;

Osservazioni

Gli esempi *dovrebbero* garantire la corretta terminazione delle attività.

Examples

Utilizzando un buffer sincronizzato

```
with Ada.Containers.Synchronized_Queue_Interfaces;
with Ada.Containers.Unbounded_Synchronized_Queues;
with Ada.Text_IO;

procedure Producer_Consumer_V1 is
   type Work_Item is range 1 .. 100;

   package Work_Item_Queue_Interfaces is
     new Ada.Containers.Synchronized_Queue_Interfaces
           (Element_Type => Work_Item);

   package Work_Item_Queues is
     new Ada.Containers.Unbounded_Synchronized_Queues
           (Queue_Interfaces => Work_Item_Queue_Interfaces);

   Queue : Work_Item_Queues.Queue;

   task type Producer;
   task type Consumer;

   Producers : array (1 .. 1)  of Producer;
   Consumers : array (1 .. 10) of Consumer;
```

```
   task body Producer is
   begin
      for Item in Work_Item loop
         Queue.Enqueue (New_Item => Item);
      end loop;
   end Producer;

   task body Consumer is
      Item : Work_Item;
   begin
      loop
         Queue.Dequeue (Element => Item);
         Ada.Text_IO.Put_Line (Work_Item'Image (Item));
      end loop;
   end Consumer;

begin
   null;
end Producer_Consumer_V1;
```

Si noti che sono stato pigro qui: non esiste una terminazione corretta delle attività del consumatore, una volta che tutti gli elementi di lavoro sono stati consumati.

Schema produttore-consumatore che utilizza il meccanismo Ada Rendezvous

Una soluzione sincrona produttore-consumatore assicura che il consumatore legga ogni singolo elemento scritto dal produttore esattamente una volta. Le soluzioni asincrone consentono al consumatore di campionare l'output del produttore. O il consumatore consuma i dati più velocemente di quanto non sia prodotto, oppure il consumatore consuma i dati più lentamente di quanto non sia stato prodotto. Il campionamento consente al consumatore di gestire i dati attualmente disponibili. Questi dati possono essere solo un campionamento dei dati prodotti o potrebbero essere già dati di consumo.

```
-----------------------------------------------------------------
-- synchronous PC using Rendezvous --
-----------------------------------------------------------------
with Ada.Text_IO; use Ada.Text_IO;

procedure PC_Rendezvous is
   task Producer;
   task Consumer is
      entry Buf(Item : in Integer);
   end Consumer;
   task body Producer is
   begin
      for I in 1..10 loop
         Put_Line("Producer writing" & Integer'Image(I));
         Consumer.Buf(I);
      end loop;
   end Producer;
   task body Consumer is
      Temp : Integer;
   begin
      loop
         select
            accept Buf(Item : in Integer) do
```

```
               temp := Item;
            end;
            Put_Line("Consumer read" & Integer'Image(Temp));
         or
            terminate;
         end select;
      end loop;
   end Consumer;

begin
   null;
end PC_Rendezvous;
```

Produttore-consumatore con un consumatore di campionamento

Questo esempio utilizza la procedura principale come attività del produttore. In Ada la procedura principale viene sempre eseguita in un'attività separata da tutte le altre attività del programma, vedere un esempio minimo .

```
------------------------------------------------------------------
-- Sampling Consumer --
------------------------------------------------------------------
with Ada.Text_IO; use Ada.Text_IO;

procedure Sampling_PC is
   protected Buf is
      procedure Write(Item : in Integer);
      function Read return Integer;
      procedure Set_Done;
      function Get_Done return Boolean;
   private
      Value : Integer := Integer'First;
      Is_Done : Boolean := False;
   end Buf;
   protected body Buf is
      procedure Write(Item : in Integer) is
      begin
         Value := Item;
      end Write;
      function Read return Integer is
      begin
         return Value;
      end Read;
      procedure Set_Done is
      begin
         Is_Done := True;
      end Set_Done;
      function Get_Done return Boolean is
      begin
         return Is_Done;
      end Get_Done;
   end Buf;

   task Consumer;
   task body Consumer is
   begin
      while not Buf.Get_Done loop
          Put_Line("Consumer read" & Integer'Image(Buf.Read));
```

```
      end loop;
   end Consumer;

begin
   for I in 1..10 loop
     Put_Line("Producer writing" & Integer'Image(I));
     Buf.Write(I);
   end loop;
   Buf.Set_Done;
end Sampling_PC;
```

Produttori e consumatori multipli Condivisione dello stesso buffer

Questo esempio mostra più produttori e consumatori che condividono lo stesso buffer. Le voci protette in Ada implementano una coda per gestire le attività in attesa. La politica di accodamento predefinita è First In First Out.

```
-----------------------------------------------------------------------
-- Multiple producers and consumers sharing the same buffer --
-----------------------------------------------------------------------
with Ada.Text_IO; use Ada.Text_Io;

procedure N_Prod_Con is
   protected Buffer is
      Entry Write(Item : in Integer);
      Entry Read(Item : Out Integer);
   private
      Value  : Integer := Integer'Last;
      Is_New : Boolean := False;
   end Buffer;
   protected body Buffer is
      Entry Write(Item : in Integer) when not Is_New is
      begin
         Value := Item;
         Is_New := True;
      end Write;
      Entry Read(Item : out Integer) when Is_New is
      begin
         Item := Value;
         Is_New := False;
      end Read;
   end Buffer;

   task type Producers(Id : Positive) is
      Entry Stop;
   end Producers;
   task body Producers is
      Num : Positive := 1;
   begin
      loop
         select
            accept Stop;
            exit;
         or
            delay 0.0001;
         end select;
         Put_Line("Producer" & Integer'Image(Id) & " writing" & Integer'Image(Num));
         Buffer.Write(Num);
```

```
         Num := Num + 1;
      end loop;
   end Producers;

   task type Consumers(Id : Positive) is
      Entry Stop;
   end Consumers;

   task body Consumers is
      Num : Integer;
   begin
      loop
         select
            accept stop;
            exit;
         or
            delay 0.0001;
         end select;
         Buffer.Read(Num);
         Put_Line("Consumer" & Integer'Image(ID) & " read" & Integer'Image(Num));
      end loop;
   end Consumers;
   P1 : Producers(1);
   P2 : Producers(2);
   P3 : Producers(3);
   C1 : Consumers(1);
   C2 : Consumers(2);
   C3 : Consumers(3);
begin
   delay 0.2;
   P1.Stop;
   P2.Stop;
   P3.Stop;
   C1.Stop;
   C2.Stop;
   C3.Stop;
end N_Prod_Con;
```

Leggi Attuazione del modello produttore-consumatore online:
https://riptutorial.com/it/ada/topic/8632/attuazione-del-modello-produttore-consumatore

Capitolo 3: Compito

Sintassi

- compito Task_Name;
- task Task_Name è Entries end;
- task body Task_Name is Dichiarazioni iniziano Code end;

Examples

Un compito semplice

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

procedure Main is
   task My_Task;
   task body My_Task is
   begin
      Put_Line ("Hello from My_Task");
   end;
begin
   Put_Line ("Hello from Main");
end;
```

Risultato

L'ordine di `Put_Line` può variare.

```
Hello from My_Task
Hello from Main
```

Un compito semplice e un ciclo

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

procedure Main is
   task My_Task;
   task body My_Task is
   begin
      for I in 1 .. 4 loop
         Put_Line ("Hello from My_Task");
      end loop;
   end;
begin
   Put_Line ("Hello from Main");
end;
```

Risultato

L'ordine di `Put_Line` può variare.

```
Hello from My_Task
Hello from Main
Hello from My_Task
Hello from My_Task
Hello from My_Task
```

Un compito semplice e due cicli

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

procedure Main is
   task My_Task;
   task body My_Task is
   begin
      for I in 1 .. 4 loop
         Put_Line ("Hello from My_Task");
      end loop;
   end;
begin
   for I in 1 .. 4 loop
      Put_Line ("Hello from Main");
   end loop;
end;
```

Risultato

L'ordine di `Put_Line` può variare.

```
Hello from My_Task
Hello from My_Task
Hello from Main
Hello from My_Task
Hello from Main
Hello from My_Task
Hello from Main
Hello from Main
```

Due semplici compiti e due loop

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

procedure Main is
   task My_Task_1;
   task My_Task_2;

   task body My_Task_1 is
   begin
```

```
      for I in 1 .. 4 loop
         Put_Line ("Hello from My_Task_1");
      end loop;
   end;

   task body My_Task_2 is
   begin
      for I in 1 .. 4 loop
         Put_Line ("Hello from My_Task_2");
      end loop;
   end;
begin
   null;
end;
```

Risultato

L'ordine di `Put_Line` può variare.

```
Hello from My_Task_1
Hello from My_Task_1
Hello from My_Task_2
Hello from My_Task_1
Hello from My_Task_2
Hello from My_Task_1
Hello from My_Task_2
Hello from My_Task_2
```

Un'attività che incrementa un numero dopo l'immissione

L'utente può chiamare `Incrementor.Increment K` numero di volte premendo un tasto all'interno di `'0' .. '9'` ed è possibile chiamare `Incrementor.Increment` più veloce del `task Incrementor` può incrementare `I`

```
with Ada.Text_IO;
with Ada.Integer_Text_IO;

procedure Main is
   use Ada.Text_IO;
   task Incrementor is
      entry Increment;
   end;
   task body Incrementor is
      use Ada.Integer_Text_IO;
      I : Integer := 0;
   begin
      loop
         accept Increment;
         I := I + 1;
         Put (I, 0);
         delay 0.1;
      end loop;
   end;
   K : Character;
begin
```

```
   loop
      Get_Immediate (K);
      if K in '0' .. '9' then
         for I in 1 .. Natural'Value (K & "") loop
            Incrementor.Increment;
         end loop;
      end if;
   end loop;
end;
```

Gestione degli interrupt

Gli interrupt sono gestiti da una procedura protetta senza parametri.

```
------------------------------------------------------------------
-- Interrupt Counting Package --
------------------------------------------------------------------
with Ada.Interrupts.Names; use Ada.Interrupts.Names;

package Ctl_C_Handling is

   protected CTL_C_Handler is
      procedure Handle_Int with
        Interrupt_Handler,
        Attach_Handler => SIGINT;
      entry Wait_For_Int;
   private
      Pending_Int_Count : Natural := 0;
   end Ctl_C_Handler;

   task CTL_Reporter is
      entry Stop;
   end CTL_Reporter;

end Ctl_C_Handling;
```

Il corpo del pacchetto mostra come funziona la procedura protetta. In questo caso un booleano non viene utilizzato nell'oggetto protetto perché gli interrupt arrivano più velocemente di quanto siano gestiti. L'attività CTL_Reporter gestisce gli interrupt ricevuti.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ctl_C_Handling; use CTL_C_Handling;
with Ada.Calendar; use Ada.Calendar;

package body Ctl_C_Handling is

   -------------------
   -- CTL_C_Handler --
   -------------------

   protected body CTL_C_Handler is

      ----------------
      -- Handle_Int --
      ----------------

      procedure Handle_Int is
```

```
      begin
         Pending_Int_Count := Pending_Int_Count + 1;
      end Handle_Int;

      ------------------
      -- Wait_For_Int --
      ------------------

      entry Wait_For_Int when Pending_Int_Count > 0 is
      begin
         Pending_Int_Count := Pending_Int_Count - 1;
      end Wait_For_Int;

   end CTL_C_Handler;

   ------------------
   -- CTL_Reporter --
   ------------------

   task body CTL_Reporter is
      type Second_Bin is mod 10;
      type History is array(Second_Bin) of Natural;

      ---------------------
      -- Display_History --
      ---------------------

      procedure Display_History(Item : History) is
         Sum : Natural := 0;
      begin
         for I in Item'Range loop
            Put_Line("Second: " & Second_Bin'Image(I) & " :" & Natural'Image(Item(I)));
            Sum := Sum + Item(I);
         end loop;
         Put_Line("Total count: " & Natural'Image(Sum));
         New_Line(2);
      end Display_History;

      One_Second_Count : Natural := 0;
      Next_Slot : Second_Bin := 0;
      Next_Second : Time := Clock + 1.0;
      Ten_Second_History : History := (Others => 0);

   begin
      loop
         Select
            Accept Stop;
            exit;
         else
            select
               CTL_C_Handler.Wait_For_Int;
               One_Second_Count := One_Second_Count + 1;
            or
               delay until Next_Second;
               Next_Second := Next_Second + 1.0;
               Ten_Second_History(Next_Slot) := One_Second_Count;
               Display_History(Ten_Second_History);
               Next_Slot := Next_Slot + 1;
               One_Second_Count := 0;
            end Select;
         end Select;
```

```
      end loop;
   end CTL_Reporter;
end Ctl_C_Handling;
```

Un esempio di programma principale per esercitare questo pacchetto è:

```
------------------------------------------------------------------
-- Ada2012 Interrupt Handler Example --
------------------------------------------------------------------
with Ada.Text_IO; use Ada.Text_IO;
with Ctl_C_Handling; use CTL_C_Handling;

procedure Interrupt01 is
begin
   Delay 40.0;
   CTL_Reporter.Stop;
   Put_Line("Program ended.");
end Interrupt01;
```

Leggi Compito online: https://riptutorial.com/it/ada/topic/7345/compito

Capitolo 4: Enumerazione

Sintassi

- function Enumeration ' Image (Argument: Enumeration'Base) return String;
- function Enumeration ' Img return String; - GNAT
- function Enumeration'Val (Argument: Universal_Integer) return Enumeration'Base;
- function Enumeration'Pos (Argument: Enumeration'Base) return Universal_Integer;
- function Enumeration'Enum_Rep (Argument: Enumeration'Base) return Universal_Integer;
- function **Literal** 'Enum_Rep return Universal_Integer; - GNAT
- funzione **Letterale** 'Indirizzo return System.Address;
- per Enumeration use (Literal_1 => **Universal_Integer** , Literal_n => **Universal_Integer**);
- (**Literal** in Enumeration) restituisce Boolean;

Examples

Iterazione di letterali

Un letterale all'interno di un'enumerazione è di tipo discreto, quindi è possibile utilizzare l'attributo `Image` per scoprire quale letterale è come una forma di testo. Si noti che questo stampa la stessa parola come nel codice (ma in maiuscolo).

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

procedure Main is
   type Fruit is (Banana, Pear, Orange, Melon);
begin
   for I in Fruit loop
      Put (Fruit'Image (I));
      New_Line;
   end loop;
end;
```

Risultato

```
BANANA
PEAR
ORANGE
MELON
```

Utilizzando il pacchetto Enumeration_IO

Invece di attribuire `Image` e `Ada.Text_IO.Put` su letterali di enumerazione, possiamo usare solo il pacchetto generico `Ada.Text_IO.Enumeration_IO` per stampare i letterali.

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

procedure Main is
   type Fruit is (Banana, Pear, Orange, Melon);
   package Fruit_IO is new Enumeration_IO (Fruit); use Fruit_IO;
begin
   for I in Fruit loop
      Put (I);
      New_Line;
   end loop;
end;
```

Risultato

```
BANANA
PEAR
ORANGE
MELON
```

Maiuscole minuscole per i caratteri maiuscoli

L' `Image` attributo capitalizza tutti i caratteri dei valori letterali di enumerazione. La funzione `Case_Rule_For_Names` applica in maiuscolo per il primo carattere e fa il resto in minuscolo.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.Maps.Constants; use Ada.Strings.Maps.Constants;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;

procedure Main is
   type Fruit is (Banana, Pear, Orange, Melon);
   function Case_Rule_For_Names (Item : String) return String is
   begin
      return Translate (Item (Item'First .. Item'First), Upper_Case_Map) & Translate (Item
(Item'First + 1 .. Item'Last), Lower_Case_Map);
   end;
begin
   for I in Fruit loop
      Put (Case_Rule_For_Names (Fruit'Image (I)));
      New_Line;
   end loop;
end;
```

Risultato

```
Banana
Pear
Orange
Melon
```

Title Case, Using Enumeration_IO, For a Subrange

Combinando il cambiamento di maiuscole / minuscole con Enumeration_IO e utilizzando un buffer di testo per l'immagine. Il primo personaggio è manipolato sul posto.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Characters.Handling; use Ada.Characters.Handling;

procedure Main is
    type Fruit is (Banana, Pear, Orange, Melon);
    package Fruit_IO is new Enumeration_IO (Fruit);
    Buffer : String (1 .. Fruit'Width);
begin
   for I in Fruit range Pear .. Fruit'Last loop
       Fruit_IO.Put (To => Buffer,
                     Item => I,
                     Set => Lower_Case);
       Buffer (Buffer'First) := To_Upper (Buffer (Buffer'First));
       Put_Line (Buffer);
   end loop;
end;
```

Risultato

```
Pear
Orange
Melon
```

Leggi Enumerazione online: https://riptutorial.com/it/ada/topic/5930/enumerazione

Capitolo 5: File e flussi I / O

Osservazioni

La libreria standard Ada fornisce I / O di file tradizionali di testo o dati binari, nonché I / O di file in streaming. I file di dati binari saranno sequenze di valori di un tipo, mentre i file di flusso possono essere sequenze di valori di tipi possibilmente diversi.

Per leggere e scrivere elementi di tipi diversi da / per lo streaming di file, Ada utilizza sottoprogrammi contraddistinti dagli attributi dei tipi, ovvero `'Read` `'Write`, `'Write` `'Input`, `'Input` `'Output` e `'Output`. Gli ultimi due leggeranno e scrivono limiti di array, discriminanti di record e tag di tipo, oltre agli input e output nudi che verranno `Read` e `'Write`.

Examples

Crea e scrivi su file

Le procedure `Create`, `Put_Line`, `Close` dal pacchetto `Ada.Text_IO` vengono utilizzate per creare e scrivere nel file `file.txt`.

```
with Ada.Text_IO;

procedure Main is
   use Ada.Text_IO;
   F : File_Type;
begin
   Create (F, Out_File, "file.txt");
   Put_Line (F, "This string will be written to the file file.txt");
   Close (F);
end;
```

File risultante `file.txt`

```
This string will be written to the file.txt
```

Crea e scrivi su un flusso

Gli attributi orientati al flusso dei sottotipi sono chiamati a scrivere oggetti in un file, nudi e usando rappresentazioni di default binarie.

```
with Ada.Streams.Stream_IO;

procedure Main is
   type Fruit is (Banana, Orange, Pear);
   type Color_Value is range 0 .. 255;
   type Color is record
```

```
      R, G, B : Color_Value;
   end record;

   Fruit_Colors : constant array (Fruit) of Color :=
     (Banana => Color'(R => 243, G => 227, B => 18),
      Orange => Color'(R => 251, G => 130, B => 51),
      Pear   => Color'(R => 158, G => 181, B => 94));

   use Ada.Streams.Stream_IO;

   F : File_Type;

begin
   Create (F, Name => "file.bin");
   for C in Fruit_Colors'Range loop
      Fruit'Write (Stream (F), C);
      Color'Write (Stream (F), Fruit_Colors (C));
   end loop;
   Close (F);
end Main;
```

File risultante

```
00000000  00 2e f3 00 e3 00 12 00  01 2e fb 00 82 00 33 00
00000010  02 2e 9e 00 b5 00 5e 00
```

Apri e leggi dal file di flusso

Leggi i dati di Crea e scrivi su un flusso di nuovo in un programma.

```
with Ada.Streams.Stream_IO;

procedure Main is
   --
   --  ... same type definitions as in referenced example
   --
   Fruit_Colors : array (Fruit) of Color;

   use Ada.Streams.Stream_IO;

   F : File_Type;
   X : Fruit;
begin
   Open (F, Mode => In_File, Name => "file.bin");
   loop
      Fruit'Read (Stream (F), X);
      Color'Read (Stream (F), Fruit_Colors (X));
   end loop;
exception
   when End_Error =>
      Close (F);
   pragma Assert  -- check data are the same
     (Fruit_Colors (Banana) = Color'(R => 243, G => 227, B => 18) and
      Fruit_Colors (Orange) = Color'(R => 251, G => 130, B => 51) and
      Fruit_Colors (Pear)   = Color'(R => 158, G => 181, B => 94));
end Main;
```

Leggi File e flussi I / O online: https://riptutorial.com/it/ada/topic/8865/file-e-flussi-i---o

Capitolo 6: Generosità in Ada

Examples

Sottoprogrammi generici

Sottoprogrammi generici sono utili per creare sottoprogrammi che hanno la stessa struttura per diversi tipi. Ad esempio, per scambiare due oggetti:

```
generic
    type A_Type is private;
procedure Swap (Left, Right : in out A_Type) is
    Temp : A_Type := Left;
begin
    Left := Right;
    Right := Temp;
end Swap;
```

Pacchetti generici

Nel pacchetto generico di Ada, dopo l'istanziazione, i dati sono duplicati; cioè, se contengono variabili globali, ciascuna istanza avrà la propria copia della variabile, correttamente digitata e indipendente dalle altre.

```
generic
    type T is private;
package Gen is
    type C is tagged record
        V : T;
    end record;
    G : Integer;
end Gen;
```

Parametri generici

Ada offre un'ampia varietà di parametri generici difficili da tradurre in altre lingue. I parametri utilizzati durante l'istanziazione e di conseguenza quelli su cui l'unità generica può fare affidamento possono essere variabili, tipi, sottoprogrammi o istanze di pacchetto, con determinate proprietà. Ad esempio, il seguente fornisce un algoritmo di ordinamento per qualsiasi tipo di array:

```
generic
    type Component is private;
    type Index is (<>);
    with function "<" (Left, Right : Component) return Boolean;
    type Array_Type is array (Index range <>) of Component;
procedure Sort (A : in out Array_Type);
```

Leggi Generosità in Ada online: https://riptutorial.com/it/ada/topic/9322/generosita-in-ada

Capitolo 7: Immagine di attributo

introduzione

Gli attributi di sottotipo `'Image` e `'Value` assumerà, rispettivamente, un valore scalare e una stringa e restituiscono, rispettivamente, una stringa e un valore scalare. Il risultato di `'Image` può essere inserito in `'Value` per ottenere il valore originale. Il contrario è anche vero.

L' `__Scalar_Object__'Image` può essere utilizzato direttamente sugli oggetti (da Ada 2012-TC-1).

Sintassi

- function Scalar'Image (Argument: Scalar'Base) return String;
- function Discrete'Image (Argument: Discrete'Base) return String;
- function Integer'Image (Argument: Integer'Base) return String;
- function Enumeration'Image (Argument: Enumeration'Base) return String;
- function Real'Image (Argument: Real'Base) return String;
- function Numeric'Image (Argument: Numeric'Base) return String;
- function Scalar'Value (Argument: String) return Scalar'Base;
- function Discrete'Value (Argument: String) return Discrete'Base;
- function Integer'Value (Argument: String) return Integer'Base;
- function Enumeration'Value (Argument: String) return Enumeration'Base;
- function Real'Value (Argument: String) restituisce Real'Base;
- function **Scalar_Object** 'Image return String;

Osservazioni

Si noti che `'Image` può sostenere risultati definiti dall'implementazione (RM 3.5), ovvero quando alcuni caratteri grafici necessari per il risultato `String` non sono definiti in `Character` . Considera i repertori più ampi di `'Wide_Image` e `'Wide_Wide_Image` .

Ada 2012 (TC-1)

L'autorizzazione per utilizzare l'attributo `__Scalar_Object__'Image` direttamente su un oggetto è stata aggiunta in Ada 2012-TC-1 (aprile 2016).

Examples

Stampa float usando l'attributo Image

Ada 2012 (TC-1)

```
with Ada.Text_IO;

procedure Main is
```

```
   type Some_Float digits 8 range 0.0 .. 10.0;
   X : Some_Float := 2.71;
begin
   Ada.Text_IO.Put_Line (X'Image);
end Main;
```

Risultato

```
2.71000E+00
```

Stampa il numero intero usando l'attributo Immagine

Ada 2012 (TC-1)

```
with Ada.Text_IO;

procedure Main is
   type Some_Integer is range -42 .. 42;
   X : Some_Integer := 17;
begin
   Ada.Text_IO.Put_Line (X'Image);
end Main;
```

Risultato

```
17
```

Stampa l'enumerazione usando l'attributo Immagine

Ada 2012 (TC-1)

```
with Ada.Text_IO;

procedure Main is
   type Fruit is (Banana, Orange, Pear);
   X : Fruit := Orange;
begin
   Ada.Text_IO.Put_Line (X'Image);
   Ada.Text_IO.Put_Line (Pear'Image);
end Main;
```

Risultato

```
ORANGE
PEAR
```

Stampa Enumerazione usando l'attributo Immagine

```
with Ada.Text_IO;

procedure Main is
   type Fruit is (Banana, Orange, Pear);
   X : Fruit := Orange;
begin
   Ada.Text_IO.Put_Line (Fruit'Image (X));
end Main;
```

Risultato

```
ORANGE
```

Stampa intero usando l'immagine attributo

```
with Ada.Text_IO;

procedure Main is
   X : Integer := 17;
begin
   Ada.Text_IO.Put_Line (Integer'Image (X));
end Main;
```

Risultato

```
17
```

Stampa Float usando l'attributo Image

```
with Ada.Text_IO;

procedure Main is
   X : Float := 2.71;
begin
   Ada.Text_IO.Put_Line (Float'Image (X));
end Main;
```

Risultato

```
2.71000E+00
```

Come Inverses

```
with Ada.Text_IO;

procedure Image_And_Value is
   type Fruit is (Banana, Orange, Pear);
   X  : Fruit := Orange;
begin
   Ada.Text_IO.Put_Line (Boolean'Image
      (Fruit'Value (Fruit'Image (X)) = X
          and
       Fruit'Image (Fruit'Value ("ORANGE")) = "ORANGE"));
end Image_And_Value;
```

Risultato

```
TRUE
```

Leggi Immagine di attributo online: https://riptutorial.com/it/ada/topic/4290/immagine-di-attributo

Capitolo 8: pacchetto Ada.Text_IO

introduzione

Il pacchetto `Ada.Text_IO` viene utilizzato per inserire testo o recuperare testo da file o console.

Examples

Put_Line

Stampa una stringa con una nuova riga.

```
with Ada.Text_IO;

procedure Put_Text is
    use Ada.Text_IO;
    S : String := "Hello";
begin
    Put_Line ("Hello");
    Put_Line (Standard_Output, "Hello");
    Put_Line (Standard_Error, "Hello error");
    Put_Line (S & " World");
end;
```

Risultato

```
Hello
Hello
Hello error
Hello World
```

Leggi pacchetto Ada.Text_IO online: https://riptutorial.com/it/ada/topic/8839/pacchetto-ada-text-io

Capitolo 9: Pacchi

Sintassi

- con Package_Name_To_Include;
- il pacchetto New_Package_Name rinomina Package_To_Rename;
- usa Package_Name;
- il pacchetto Parent_Name.Child_Name è

Osservazioni

Il pacchetto fornisce:

- Codice incapsulamento
- Compilazione separata
- Nascondi procedure, funzioni, operatori su tipi privati

Somiglianze o analoghi in altre lingue:

- Spazio dei nomi C ++
- Pacchetti Java

Examples

Maggiori informazioni sui pacchetti

In Hello World , sei stato introdotto nel pacchetto `Ada.Text_IO` e come usarlo per eseguire operazioni di I / O all'interno del tuo programma. I pacchetti possono essere ulteriormente manipolati per fare molte cose diverse.

Rinominare : per rinominare un pacchetto, si utilizza la parola chiave `renames` in una dichiarazione di pacchetto, in quanto tale:

```
package IO renames Ada.Text_IO;
```

Ora, con il nuovo nome, puoi usare la stessa notazione `Put_Line` per funzioni come `Put_Line` (cioè `IO.Put_Line`), o puoi semplicemente `use` con l' `use IO` . Ovviamente, `use IO` o `IO.Put_Line` userai le funzioni del pacchetto `Ada.Text_IO` .

Visibilità e isolamento : nell'esempio *Hello World* abbiamo incluso il pacchetto Ada.Text_IO utilizzando una clausola `with` . Ma abbiamo anche dichiarato che volevamo `use Ada.Text_IO` sulla stessa riga. La dichiarazione `use Ada.Text_IO` potrebbe essere stata spostata nella parte dichiarativa della procedura:

```
with Ada.Text_IO;

procedure hello_world is
   use Ada.Text_IO;
begin
   Put_Line ("Hello, world!");
end hello_world;
```

In questa versione, le procedure, le funzioni e i tipi di `Ada.Text_IO` sono direttamente disponibili all'interno della procedura. Al di fuori del blocco in cui viene utilizzato `Ada.Text_IO` , dovremmo utilizzare la notazione `Ada.Text_IO` per invocare, ad esempio:

```
with Ada.Text_IO;

procedure hello_world is
begin
   Ada.Text_IO.Put ("Hello, ");       --  The Put function is not directly visible here
   declare
      use Ada.Text_IO;
   begin
      Put_Line ("world!");            --  But here Put_Line is, so no Ada.Text_IO. is needed
   end;
end hello_world;
```

Questo ci permette di isolare l'uso ... delle dichiarazioni dove sono necessarie.

Relazione padre-figlio

Come un modo per suddividere i programmi Ada, i pacchetti possono avere i cosiddetti bambini. Anche questi possono essere pacchetti. Un pacchetto figlio ha un privilegio speciale: può vedere le dichiarazioni nella parte privata del pacchetto genitore. Un uso tipico di questa visibilità speciale è quando si forma una gerarchia di tipi derivati nella programmazione orientata agli oggetti.

```
package Orders is
   type Fruit is (Banana, Orange, Pear);
   type Money is delta 0.01 digits 6;

   type Bill is tagged private;

   procedure Add
     (Slip   : in out Bill;
      Kind   : in     Fruit;
      Amount : in     Natural);

   function How_Much (Slip : Bill) return Money;

   procedure Pay
     (Ordered : in out Bill;
      Giving  : in     Money);

private
   type Bill is tagged record
      --  ...
      Sum : Money := 0.0;
   end record;
end Orders;
```

Qualsiasi unità Ada guidata da `with Orders;` può dichiarare oggetti di tipo `Bill` e quindi chiamare operazioni `Add` , `How_Much` e `Pay` . Tuttavia, non vede i componenti di `Bill` e nemmeno di `Orders.Bill` , poiché la definizione completa del tipo è nascosta nella parte **privata** degli `Orders` . La definizione completa non è nascosta dai pacchetti figlio, tuttavia. Questa visibilità facilita l'estensione del tipo, se necessario. Se un tipo viene dichiarato nel pacchetto figlio come derivato da `Bill` , questo tipo di ereditarietà può manipolare direttamente i componenti di `Bill` .

```
package Orders.From_Home is
   type Address is new String (1 .. 120);

   type Ordered_By_Phone is new Bill with private;

   procedure Deliver
     (Ordered : in out Ordered_By_Phone;
      Place   : in     Address);

private
   type Ordered_By_Phone is new Bill with
      record
         Delivered : Boolean := False;
         To        : Address;
      end record;
end Orders.From_Home;
```

`Orders.From_Home` è un pacchetto figlio di `Orders` . Type `Ordered_By_Phone` è derivato da `Bill` e include il suo componente record `Sum` .

Leggi Pacchi online: https://riptutorial.com/it/ada/topic/7322/pacchi

Capitolo 10: Tipi parametrizzati

introduzione

Tutti i tipi compositi diversi dagli array possono avere discriminanti, che sono componenti con proprietà speciali. I discriminanti possono essere di tipo discreto o di accesso. In quest'ultimo caso il tipo di accesso può essere un tipo di accesso con nome o può essere anonimo. Una discriminante di un tipo di accesso anonimo è chiamata discriminante di accesso per analogia con un parametro di accesso.

Examples

Tipi di record discriminati

Nel caso di un tipo di record discriminato, alcuni componenti sono noti come discriminanti e i componenti rimanenti possono dipendere da questi. Le discriminanti possono essere pensate come parametrizzare il tipo e la sintassi rivela questa analogia. In questo esempio creiamo un tipo che fornisce una matrice quadrata con un parametro positivo come:

```
type Square(X: Positive) is
    record
        S: Matrix(1 .. X, 1 .. X);
    end record;
```

Quindi per creare un quadrato di 3 per 3, chiama semplicemente il tipo quadrato in questo modo:

```
Sq: Square(3);
```

Strutture record varianti

Un discriminante di un tipo di record può influenzare la struttura degli oggetti. Una scelta di componenti può esistere in un oggetto in base al fatto che un discriminante aveva avuto un valore particolare quando l'oggetto è stato creato. Per supportare questa variazione, la definizione di un tipo di record include una distinzione per casi che dipendono dalla discriminante:

```
type Fruit is (Banana, Orange, Pear);

type Basket (Kind : Fruit) is
   record
      case Kind is
         when Banana =>
            Bunch_Size      : Positive;
            Bunches_Per_Box : Natural;
         when Pear | Orange =>
            Fruits_Per_Box  : Natural;
      end case;
   end record;
```

Quindi per creare una scatola per le banane,

```
Box : Basket (Banana);
```

La `Box` oggetto ha ora due componenti di registrazione in aggiunta alla sua discriminante, `Kind` , cioè `Bunch_Size` e `Bunches_Per_Box` .

Leggi Tipi parametrizzati online: https://riptutorial.com/it/ada/topic/9311/tipi-parametrizzati

Capitolo 11: Tipi scalari

introduzione

Nella gerarchia dei tipi di Ada, i tipi elementari hanno serie di valori logicamente indivisibili. Tra questi tipi vi sono i tipi di accesso (tipi di puntatore) e i tipi scalari. I tipi scalari possono essere classificati come *enumerazione* , *carattere* e *numerico* . Questi tipi formano l'argomento di questo argomento. Oltre agli insiemi di valori, i tipi hanno un insieme di operazioni applicabili ai rispettivi scalari, come ad esempio *successore* o `"+"` .

Sintassi

1. **tipo** ... **è** ...

Parametri

ellissi	Che cosa
... (1)	per ricevere il nome del tipo
... (2)	per ricevere le caratteristiche del tipo usando le parole chiave: **delta** , **cifre** , **intervallo**

Osservazioni

Tutte le definizioni di tipo scalare, tranne l'enumerazione e gli interi modulari possono includere un vincolo di **intervallo** .

Un vincolo di intervallo specifica un limite inferiore e un limite superiore dell'insieme di valori da includere nel tipo. Per i tipi di punti fissi, specificare un intervallo è obbligatorio: i valori di questi tipi saranno intesi come multipli di una piccola frazione di due, ad esempio di $1/2^5$. Più piccole diventano queste frazioni, più precisa è la rappresentazione, al costo dell'intervallo che può essere rappresentato utilizzando i bit disponibili.

Ulteriori aspetti delle definizioni di tipo possono essere forniti, come una `Size` desiderata in bit e altri elementi di rappresentazione. Ada 2012 aggiunge aspetti della programmazione basata su contratto come `Static_Predicate` .

Examples

Enumerazione

```
type Fruit is (Banana, Orange, Pear);
```

```
Choice : Fruit := Banana;
```

Un tipo di carattere è un'enumerazione che include un carattere letterale:

```
type Roman_Numeral is
    ('I', 'V', 'X', 'L', 'C', 'D', 'M', Unknown);`
```

Integer cantato

```
type Grade is range 0 .. 15;

B   : Grade := 11;
C   : Grade := 8;
Avg : Grade := (B + C) / 2;  -- Avg = 9
```

Intero modulare

Questi sono i tipi "un po 'giocherellona". Hanno anche degli operatori logici, come **xor** , e si "avvolgono" al limite superiore, di nuovo a 0.

```
type Bits is mod 2**24;

L : Bits := 2#00001000_01010000_11001100# or 7;
```

Virgola mobile

Un tipo a virgola mobile è caratterizzato da cifre (decimali) che indicano la precisione minima richiesta.

```
type Distance is digits 8;

Earth : Distance := 40_075.017;
```

Punto fisso (ordinario)

Una definizione del tipo di punto fisso specifica un *delta* e un intervallo. Insieme, descrivono in che modo i valori reali dovrebbero essere approssimati in quanto sono rappresentati da potenze di due, non usando hardware in virgola mobile.

```
Shoe_Ounce : constant := 2.54 / 64.0;
type Thickness is delta Shoe_Ounce range 0.00 .. 1.00;

Strop : Thickness := 0.1;  -- could actually be 0.09375
```

Punto fisso (decimale)

I tipi decimali a virgola fissa vengono generalmente utilizzati nella contabilità. Sono caratterizzati

sia da un *delta* che da un numero di cifre decimali. Le loro operazioni aritmetiche riflettono le regole della contabilità.

```
type Money is delta 0.001 digits 10;

Oil_Price : Money := 56.402;
Loss      : Money := 0.002 / 3; -- is 0.000
```

Leggi Tipi scalari online: https://riptutorial.com/it/ada/topic/9297/tipi-scalari

Capitolo 12: Uscita di numeri

introduzione

I pacchetti standard di Ada forniscono l'output di tutti i tipi numerici. Il formato di output può essere regolato in molti modi.

Osservazioni

Nota come ogni volta che un pacchetto generico viene istanziato con un tipo numerico. Inoltre, ci sono entrambi i valori di default da impostare per l'intera istanza, e anche i modi per sovrascrivere la `Width` , ad esempio, quando si chiama `Put` con questo parametro.

Examples

Stampa interi, generosamente usando lo spazio

Le istanze di `Integer_IO` hanno una variabile di impostazione `Default_Width` che prende il numero di caratteri che ciascun numero di uscita impiega.

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

procedure Print_Integer is
    subtype Count is Integer range -1_000_000 .. 1_000_000;

    package Count_IO is new Integer_IO (Count);
    X : Count;
begin
    Count_IO.Default_Width := 12;

    X := Count'First;
    while X < Count'Last loop
        Count_IO.Put (X);
        Count_IO.Put (X + 1);
        New_Line;

        X := X + 500_000;
    end loop;
end Print_Integer;
```

Risultato

```
    -1000000
     -500000
           0
      500000
```

Stampa numeri interi, utilizzando la base 16 (esadecimale)

Una variabile di impostazioni `Default_Base` è impostata `Ada.Text_IO.Integer_IO` di `Ada.Text_IO.Integer_IO` ; inoltre, `Default_Width` è impostato in modo che l'output non possa avere spazio iniziale.

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

procedure Print_Hex is
    subtype Count is Integer range -1_000_000 .. 1_000_000;

    package Count_IO is new Integer_IO (Count);
    X : Count;
begin
    Count_IO.Default_Width := 1;
    Count_IO.Default_Base := 16;

    X := Count'First;
    while X < Count'Last loop
        Count_IO.Put (X);
        New_Line;

        X := X + 500_000;
    end loop;
end Print_Hex;
```

Risultato

```
-16#F4240#
-16#7A120#
16#0#
16#7A120#
```

Stampa numeri decimali a virgola fissa, ovvero denaro

`Ada.Text_IO.Editing` offre la formattazione di valori decimali a virgola fissa usando "stringhe di immagini". Questi descrivono l'output usando caratteri "magici" per separatori, segni di valuta, ecc.

```
with Ada.Text_IO.Editing;   use Ada.Text_IO;

procedure Print_Value is

    Max_Count      : constant := 1_000_000;

    type Fruit is (Banana, Orange, Pear);
    subtype Count is Integer range -Max_Count .. +Max_Count;

    type Money is delta 0.001 digits 10;

    package Fruit_IO is new Enumeration_IO (Fruit);
    package Money_IO is new Editing.Decimal_Output
      (Money,
       Default_Currency => "CHF",
```

```
       Default_Separator => ''');

    Inventory : constant array (Fruit) of Count :=
      (Banana => +27_420,
       Orange => +140_600,
       Pear   => -10_000);

    Price_List : constant array (Fruit) of Money :=
      (Banana => 0.07,
       Orange => 0.085,
       Pear   => 0.21);

    Format : constant Editing.Picture :=
      Editing.To_Picture ("<###BZ_ZZZ_ZZ9.99>");
begin
    Fruit_IO.Default_Width := 12;

    for F in Inventory'Range loop
        Fruit_IO.Put (F);
        Put          (" | ");
        Money_IO.Put (Item => Inventory (F) * Price_List (F),
                      Pic => Format);
        New_Line;
    end loop;
end Print_Value;
```

Risultato

```
BANANA       |  CHF     1'919.40
ORANGE       |  CHF    11'951.00
PEAR         | (CHF     2'100.00)
```

Stampa più elementi su un'unica riga

Combina le istanze dei pacchetti `_IO` , usa quella giusta con il suo tipo numerico.

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

procedure Print_Inventory is
    type Fruit is (Banana, Orange, Pear);
    subtype Count is Integer range -1_000_000 .. 1_000_000;

    package Fruit_IO is new Enumeration_IO (Fruit);
    package Count_IO is new Integer_IO (Count);

    Inventory : constant array (Fruit) of Count :=
      (Banana => 27_420,
       Orange => 140_600,
       Pear   => -10_000);

begin
    Fruit_IO.Default_Width := 12;

    for F in Inventory'Range loop
        Fruit_IO.Put (F);
        Put          (" | ");
```

```
        Count_IO.Put (Inventory (F));
        New_Line;
    end loop;
end Print_Inventory;
```

Risultato

```
BANANA       |    27420
ORANGE       |   140600
PEAR         |   -10000
```

Leggi Uscita di numeri online: https://riptutorial.com/it/ada/topic/8940/uscita-di-numeri

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Ada	B98, Community, Jacob Sparre Andersen, Jaken Herman, Jossi , manuBriot, trashgod
2	Attuazione del modello produttore-consumatore	Jacob Sparre Andersen, Jim Rogers, Jossi
3	Compito	Jim Rogers, Jossi
4	Enumerazione	B98, Jossi, Simon Wright
5	File e flussi I / O	B98, Jossi
6	Generosità in Ada	Aznhar, B98
7	Immagine di attributo	B98, Jacob Sparre Andersen, Jossi
8	pacchetto Ada.Text_IO	Jossi
9	Pacchi	B98, Jaken Herman, Jossi
10	Tipi parametrizzati	Aznhar, B98
11	Tipi scalari	B98
12	Uscita di numeri	B98