Università degli Studi di Bologna
Sede di Cesena

Corso di Laurea in Scienze dell'Informazione

Anno Accademico 1997/98

ADA

Stefano Clemente
clemente@csr.unibo.it

Caratteristiche del Linguaggio

ADA è stato progettato per risolvere le molte esigenze della programmazione real time e per questo motivo è un linguaggio molto ricco.

Le sue caratteristiche principali sono:

Leggibilità

La notazione del linguaggio rende facile la lettura dei programmi.

Tipologia Controllata

Attraverso i tipi di dato è possibile controllare che un "oggetto" assuma valori solo in un dato insieme evitando che esso assuma valori errati da un punto di vista logico.

Possibilità di Programmazione su Vasta Scala

Permette di scrivere programmi:

- ??senza limiti di dimensioni,
- ??trasportabili su più macchine,
- ??sui quali è possibile eseguire una facile manutenzione

ed a tal fine offre meccanismi per :

- ??la protezione dei dati,
- ??la compilazione separata,
- ??gestione delle librerie.

Gestione delle Segnalazioni di Eccezioni

Quando si verifica un errore, le azioni da intraprendere possono essere specificate direttamente nella sezione del programma che ha generato l'errore.

Rappresentazione dei Dati in Forma Astratta

I dati vengono rappresentati in maniera indipendente dalle operazioni logiche che su di essi vengono eseguite, rendendo così i programmi facilmente trasportabili su macchine diverse e facilmente leggibili.

Gestione Lavori (Gestione Processi)

ADA dispone di meccanismi per l'esecuzione parallela (o concorrente) di più attività (***no system calls***).

Unità di Programma Generalizzate

È possibile raggruppare in librerie parti di programma che sono di utilità generale, oppure che sono soggette a continue modifiche indipendenti dal resto del programma.

OO-Programming ed Ereditarietà

Supporto ben definito per l'interfaccia verso altri linguaggi come C, Fortran e COBOL.

Presentazione del Linguaggio

Tra le caratteristiche richieste ad un linguaggio di programmazio-ne c'è sicuramente quella di poter riutilizzare facilmente porzioni di altri programmi esistenti. Per fare ciò bisogna poter strutturare il programma in moduli detti ***librerie*** in modo tale che ogni libreria raccolga le funzionalità relative ad un certo argomento (es.: funzioni trigonometriche, funzioni per l'I/O, funzioni per la gestione di un particolare dispositivo, ecc.).
In ADA dette librerie vengono definite unità di programma generalizzate o ***packages***.

Il generico programma ADA è una procedura che richiama procedure di unità di programma generalizzate sotto forma di sottoprogrammi (funzioni, procedure) o di pacchetti applicativi (packages). Inoltre ogni programma è a sua volta un'unità di programma generalizzabile.

Esempio

Versione 1

```
with SQRT, SIMPLE_IO;
procedure PRINT_ROOT is
begin
    -- radice quadrata di 2.5
    SIMPLE_IO.PUT(SQRT(2.5));
end PRINT_ROOT;
```

Versione 2

```
with SQRT, SIMPLE_IO;
procedure PRINT_ROOT is
    use SIMPLE_IO;
begin
    -- radice quadrata di 2.5
    PUT(SQRT(2.5));
end PRINT_ROOT;
```

Versione 3

```
with SQRT, SIMPLE_IO;
procedure PRINT_ROOT is
    use SIMPLE_IO;
    X : FLOAT;
begin
    -- radice quadrata di un numero inserito
    -- dall'utente
    GET(X);
    PUT(SQRT());
end PRINT_ROOT;
```

La struttura di un programma ADA è a questo punto un po' più chiara:

```
[with <lista_nomi_packages>;]
procedure <nome>(<lista_parametri>) is
    -sequenza di dichiarazioni-
begin
    -sequenza di comandi-
[exception
    -gestori delle eccezioni-]
end [<nome>];
```

Versione 4

```
with SQRT, SIMPLE_IO;
procedure PRINT_ROOT is
    use SIMPLE_IO;
    X : FLOAT;
begin
    -- ciclo per il calcolo della radice quadrata di
    -- un numero inserito dall'utente con test sulla
    -- positività del numero
    PUT("Radici di numeri vari.");
    NEW_LINE(2);
    loop
        GET(X);
        exit when X = 0.0;
        PUT("La Radice di ");
        PUT(X);
        PUT(" è ");
        if X < 0.0 then
            PUT(" non calcolabile");
        else
            PUT(SQRT(X));
        end if;
        NEW_LINE;
    end loop;
    NEW_LINE;
    PUT("Programma finito.");
    NEW_LINE;
end PRINT_ROOT;
```

Si osservi come la procedura "`NEW_LINE`" viene richiamata una volta passandole l'argomento "`2`", mentre le altre volte senza parametri. Si tratta di due procedure diverse, ma aventi lo stesso nome ed il compilatore riconosce di volta in volta quale delle due deve applicare.

Definizione

La possibilità di assegnare lo stesso nome a oggetti diversi è detta ***overloading*** (***sovraccarico***).

In questo caso abbiamo inserito del codice per prevenire gli errori dovuti al tentativo di calcolare la radice quadrata di un numero negativo.
Supponiamo che "SQRT" ritorni un'eccezione: in questo caso se non scrivessimo niente l'APSE comunicherebbe un errore nel programma. Dovremmo allora scrivere:

Versione 5

```
with SQRT, SIMPLE_IO;
procedure PRINT_ROOT is
    use SIMPLE_IO;
    X : FLOAT;
begin
    -- ciclo per il calcolo della radice quadrata di
    -- un numero inserito dall'utente con gestore
    -- delle eccezioni
    PUT("Radici di numeri vari.");
    NEW_LINE(2);
    loop
        GET(X);
        exit when X = 0.0;
        PUT("La Radice di ");
        PUT(X);
        PUT(" è ");
        begin
            PUT(SQRT(X));
        exception
            when NUMERIC_ERROR =>
                PUT("non calcolabile");
        end;
        NEW_LINE;
    end loop;
    NEW_LINE;
    PUT("Programma finito.");
    NEW_LINE;
end PRINT_ROOT;
```

Vediamo adesso come potrebbe essere "SQRT":

```
function SQRT (F:FLOAT) return FLOAT is
    R : FLOAT;
begin
    -- calcola SQRT(F) in R
    return R;
end SQRT;
```

Il package "SIMPLE_IO" è così definito:

```
package SIMPLE_IO is
    procedure GET(F: out FLOAT);
    procedure PUT(F: in FLOAT);
    procedure PUT(S: in STRING);
    procedure NEW_LINE(N: in INTEGER:=1);
end SIMPLE_IO;
with INPUT_OUTPUT;
package body SIMPLE_IO is
    .....
    procedure GET(F: out FLOAT) is
        .....
    begin
        .....
    end GET;
    .....
end SIMPLE_IO;
```

Funzioni e Procedure sono ***sottoprogrammi***.

Bibliografia

?? "*Tutorial ADA in linea*" - http://www.scism.sbu.ac.uk/law/lawhp.html

?? **John G. P. Barnes** - "*Programmare in ADA*" - Ed. Zanichelli - (ADA83).

?? "Manuale ADA95" - http://www.adahome.com/rm95

Tipi di Dato

I tipi di dato di base di ADA estendono di poco quelli disponibili nel PASCAL.

Il vero punto di forza di ADA è (come si vedrà) nella potenza dei meccanismi per la definizione di nuovi tipi.

Il concetto di tipo di dato primitivo è leggermente differente da quello del PASCAL. I tipi predefiniti nel package STANDARD, come ad esempio BOOLEAN, CHARACTER, REAL, INTEGER e STRING, sono definiti in base a tipi più primitivi come array, enumerazioni e record.

Dichiarazioni e Assegnamenti

Gli oggetti ai quali è possibile assegnare un valore sono ***variabili*** e ***costanti***.
Per poter assegnare un valore ad un oggetto occorre innanzitutto che quest'ultimo venga dichiarato; la dichiarazione consiste nello scrivere:

```
dichiarazione ::= lista_nomi: [constant] nome_tipo [:= valore_iniziale];
lista_nomi ::= nome_oggetto , lista_nomi | nome_oggetto
```

Esempi

```
I : INTEGER;
J : INTEGER := 60;
P, Q, R : INTEGER := 100;
SALUTO : constant STRING = "ciao";
PI : constant REAL := 3.14159;
PI : constant := 3.14159; (dichiarazione di numero)
```

Come in PASCAL l'istruzione di assegnamento è:

```
variabile := espressione;
```

Esempi

```
    I := 10;
    I := P + Q;
```

Blocchi, “Scope” e Visibilità delle Variabili

Un blocco è una parte di codice del programma contenente sia dichiarazioni, sia istruzioni.

La sintassi di un blocco è:

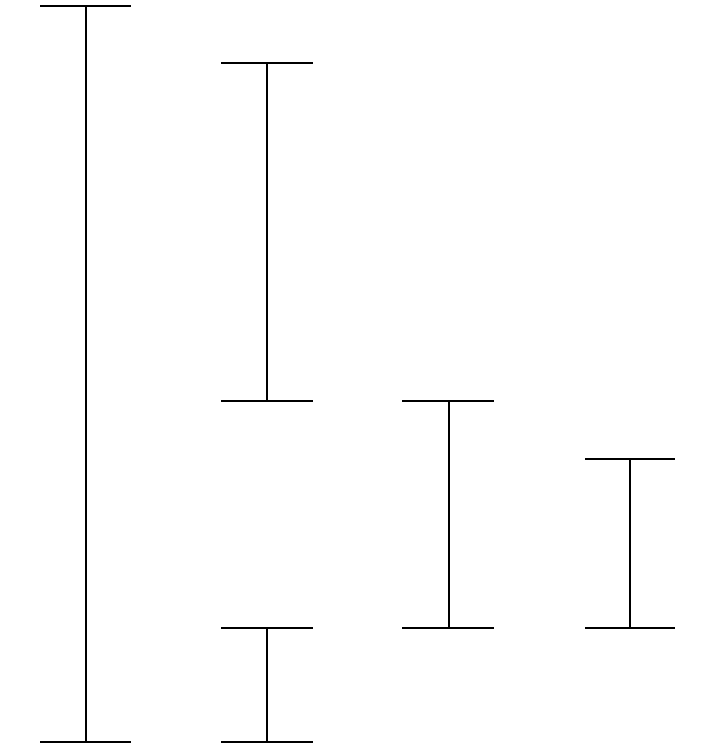
```
istruzioni ::= .... | blocco | ....
blocco ::=   declare
                 dichiarazioni
             begin
                 istruzioni
             end;
```

Un blocco viene elaborato, come qualsiasi altra istruzione, quando viene incontrato durante l’esecuzione di un programma: dapprima vengono eseguite le dichiarazioni delle variabili che vengono utilizzate nel blocco (parte tra **`declare`** e **`begin`**) e poi vengono eseguite le relative istruzioni (parte tra **`begin`** e **`end`**).
Gli oggetti della parte dichiarativa hanno vita dal momento in cui vengono dichiarati fino a quando si arriva all’**`end`** del blocco al quale appartengono.

All’interno di un blocco è possibile dichiarare variabili con lo stesso nome di variabili appartenenti a blocchi più esterni. L’effetto è quello di rendere temporaneamente invisibile la variabile più esterna.

Esempio

```
declare
    I : INTEGER:=0;
begin
    ....
    I := 100
    ....
    declare
        K : INTEGER := I;
        I : INTEGER := 0;
    begin
        ....
    end;
        ....
end;
```



Tipi e Sottotipi

Un tipo va dichiarato inserendo nella parte dichiarazioni di un blocco:

```
type nome_tipo is definizione_tipo;
```

Esempi

```
type INTEGER is ...;
type COLORI is (ROSSO, VERDE, BLU);
```

Dato un tipo, possiamo anche definire un ***sottotipo***, vale a dire un sottoinsieme dei dati appartenenti ad esso. Il sottotipo eredita tutte le operazioni possibili sugli elementi del tipo principale.

Esempio

```
subtype GIORNO_NUM is INTEGER range 1..31;
D : DAY_NUMBER;
...
I : INTEGER;
...
D := I;
...
I := D;
```

Quando vengono violate le restrizioni imposte sul tipo principale, viene generata l'eccezione CONSTRAINT_ERROR.

Si possono imporre delle restrizioni anche nella dichiarazione di una variabile.

Esempio

```
subtype GIORNO_NUM is INTEGER;
D : INTEGER range 1..31;
```

Si possono definire sottotipi di altri sottotipi.

Esempio

```
subtype GIORNI_FEB is GIORNO_NUM range 1..29;
```

Tipi Numerici

Interi

Nel package STANDARD di ADA esiste il tipo predefinito INTEGER. Vi possono poi essere, a seconda del particolare tipo di compilatore, anche i tipi SHORT_INTEGER e/o LONG_INTEGER, POSITIVE.

Per conoscere l'intervallo di interi disponibile in una realizzazione di ADA si possono consultare le due costanti predefinite SYSTEM.MIN_INT e SYSTEM.MAX_INT. Il linguaggio fornisce le normali operazioni aritmetiche sugli interi con notazione infissa.

Reali a virgola mobile

Nel package STANDARD di ADA esiste il tipo predefinito FLOAT. Vi possono poi essere, a seconda del particolare tipo di compilatore, anche i tipi SHORT_ FLOAT e/o LONG_ FLOAT.

Come per gli interi è possibile dichiarare dei tipi basati sul tipo FLOAT; ad esempio

```
type RISULTATO is digits 10 range -1.0E12..1.0E12;
```

dichiara un tipo a virgola mobile con precisione di 10 cifre e intervallo di variazione da -10^{12} a 10^{12}.

Esistono per i tipi a virgola mobile diversi ***attributi*** (= un attributo designa un valore o una proprietà di tipi, strutture, hardware ecc.). Dato un tipo a virgola mobile `T`:

`T'DIGITS`	numero di cifre decimali dichiarate per `T`;
`T'MANTISSA`	numero di bits della mantissa necessari per il numero di cifre decimali dichiarato;
`T'EMAX`	massimo valore dell'esponente nell'intervallo dichiarato.

Il poter disporre di questi attributi e di altri che qui non sono stati ricordati, permette di scrivere programmi che controllano se alcune delle caratteristiche della macchina sulla quale stanno girando possono influenzare il risultato di particolari operazioni in virgola mobile che devono eseguire.
Ada dispone delle normali operazioni sui reali (esponenziazione compresa).

Reali a virgola fissa

Per dichiarare un tipo a virgola fissa bisogna definire la distanza tra due numeri consecutivi, cioè il cosiddetto ***valore delta***:

```
type NUMERO_FIXED is delta 0.01 range -100.0..100.0;
```

vuol dire che `NUMERO_FIXED` individua tutti i reali aventi due cifre decimali e compresi tra `-100.0` e `100.0`.
ADA fornisce le operazioni in virgola fissa.

Tipi Enumerativi

In maniera simile al PASCAL è possibile definire un tipo elencando gli elementi che ne fanno parte.

Esempio

```
type COLORI is (ROSSO, VERDE, BLU);
```

Per le enumerazioni è obbligatoria la definizione di un tipo, vale a dire che non si può definire una variabile:

```
COLORE : (ROSSO, VERDE, BLU);
```

La rappresentazione interna delle enumerazioni è per valori posizionali; nel nostro esempio questo si traduce come segue:

```
ROSSO = 0
VERDE = 1
BLU   = 2
```

Questa rappresentazione permette di usare gli operatori di relazione “=”, “>=”, “<>”, “<=”.

Inoltre è possibile sovraccaricare i nomi dei valori delle enumerazioni.

Esempio

```
type TEMPERA is (BIANCO, NERO, ROSSO, GIALLO, BLU);
type RGB is (ROSSO, VERDE, BLU);
```

Sarà il compilatore a dover risolvere all'occorrenza il sovraccarico. Il programmatore può aiutare il compilatore a risolvere il sovraccarico indicando a quale valore di enumerazione si riferisce. A questo scopo può aggiungere al nome del valore il nome del tipo.

Esempio

```
TEMPERA(BLU);
RGB(BLU);
```

Tipi Enumerativi: Caratteri e Booleani

Due tipi predefiniti nel package STANDARD, sono stati definiti per enumerazione: i BOOLEAN ed i CHARACTER.

Esempio

```
type BOOLEAN is (FALSE, TRUE);
type CHARACTER is (<caratteri ASCII>);
```

Per quanto riguarda i booleani vengono fornite, oltre alle operazioni sulle enumerazioni, anche le operazioni sui booleani:
"AND", "OR", "XOR", "AND THEN", "OR ELSE".

Vettori ed Array

Un array può avere:
??un numero qualsiasi di dimensioni,
??intervallo di variazione degli indici arbitrario,
??elementi di qualsiasi tipo.

Esempio

```
TABLE : array (1..10, 1..20) of FLOAT;
TABLE(10,15) := 1.4;
```

Si può definire un tipo array. ADA permette di non specificare l'esatto intervallo di variazione delle dimensioni al tempo della dichiarazione del tipo; detto intervallo potrà essere specificato in sede di dichiarazione di una variabile di quel tipo.

Esempio

```
type BITMAP is
    array (POSITIVE range <>, POSITIVE range <>) of
    GRAY_LEVEL;
FOTOGRAFIA : BITMAP(1024,768);
```

L'intervallo di variazione degli indici di un array può anche essere un'enumerazione.

Esempio

```
type GIORNO is (LUN, MAR, MER, GIO, VEN, SAB, DOM);
type GIORNO_LAVORATIVO is array (GIORNO) of
    BOOLEAN := (LUN..VEN => TRUE, others => FALSE;
```

È inoltre possibile definire con delle variabili gli estremi dell'intervallo di variazione degli indici dell'array. Quando si incontra nel flusso dell'elaborazione la dichiarazione

```
FOTOGRAFIA : array (1..X, 1..Y) of GRAY_LEVEL;
```

gli intervalli di variazione verranno stabiliti dinamicamente in base ai valori di `X` e di `Y`.

Attraverso gli indici di un vettore si può selezionare una porzione dello stesso.

Esempio

```
type BITMAP is
```

```
    array (POSITIVE range <>, POSITIVE range <>) of
    GRAY_LEVEL;
FOTOGRAFIA : BITMAP(1024,768);
FOTOGRAFIA(1..100,1..100);
```

Infine, si può inizializzare un array al momento della sua dichiarazione.

Esempio

```
POTENZE_DI_DUE : array (1..6) of INTEGER := (2,4,8,16,32,64);
```

Stringhe

Sono un tipo predefinito nel package `STANDARD`:

```
type STRING is
    array (POSITIVE range <>) of CHARACTER;
```

Per dichiarare una stringa si scriverà

```
NOME : STRING(1..30);
```

Sulle stringhe sono definite le operazioni sui vettori, l'operatore di concatenazione "`&`" e gli operatori relazionali che operano sull'ordinamento stabilito in base all'ordinamento del tipo enumerato `CHARACTER`.

Records Semplici

La sintassi per la definizione di un record è:

```
record STRING is
    <nome_campo> : <tipo_campo>;
    ...
    <nome_campo> : <tipo_campo>;
end record;
```

Per dichiarare una variabile record occorre obbligatoriamente dichiarare prima il tipo.

Esempio

```
type DATA is
    record
        GIORNO : INTEGER range 1..31 := 12;
        MESE : STRING(1..3) := "GEN";
        ANNO : INTEGER range 1950..2050 := 1998;
    end record;
OGGI : DATA;
```

Se il campo di un record R_1 è a sua volta un record di tipo R_2, la dichiarazione di R_2 deve essere stata fatta prima della dichiarazione di R_1 (e quindi fuori da questa).

La variabile dichiarata assume per default i valori iniziali stabiliti nella dichiarazione di tipo. In sede di dichiarazione della variabile record si possono assegnare valori diversi scrivendo:

```
DOMANI : DATA(13,"GEN",1998);
DOPODOMANI : DATA(GIORNO=>14,MESE=>"GEN",ANNO=>1998);
```

Analogamente si possono eseguire degli assegnamenti:

```
DOMANI := (13,"GEN",1998);
DOPODOMANI := (GIORNO=>14,MESE=>"GEN",ANNO=>1998);
```

I campi di una variabile record possono essere riferiti, come in PASCAL, attraverso il punto; scriveremo allora:

```
DOMANI.GIORNO := 13;
```

Records con discriminanti

Questo è un primo modo per definire records che tutto sommato hanno struttura simile anche se non perfettamente identica.

Se si vuole definire un record con campi arrays aventi dimensioni di lunghezza variabile, bisogna parametrizzare il record rispetto alla lunghezza.

Esempio: si vuole definire un tipo record per memorizzare gli incassi giornalieri di un negozio.

```
type MESE (LUNGH : INTEGER range 1..31) is
    record
        NOME : STRING(1..3);
        INCASSI : array 1..LUNGH of FLOAT;
    end record;
GENNAIO : MESE(31);
FEBBRAIO : MESE(28);
```

Il parametro del tipo record è detto ***discriminante***.

Records variabili

In questo caso le strutture dei record cambiano l'una dall'altra, nel senso che i campi che compongono il record sono diversi a seconda della particolare dichiarazione.

Per definire un record variabile (o con varianti), bisogna innanzitutto prevedere un discriminante. La sintassi è la seguente:

```
type <nome_record> (<nome_discr> : <tipo_discr>) is
    record
        <nome_campo> : <tipo_campo>;
        ...
        <nome_campo> : <tipo_campo>;
        case <nome_discr> is
            when <valore_discr> => <lista_campi>;
            when <valore_discr> => <lista_campi>;
            ...
            when others => <lista_campi>;
    end record;
<lista_campi>::=<nome_campo>:<tipo_campo>;<lista_campi>|
                NULL
```

Esempio:

```
type PAGA is (ORARIA, STIPENDIO);
type IMPIEGATO (TIPO_PAGA : PAGA) is
    record
        NOME : STRING(1..30);
        ETA : INTEGER range 15..100;
        case TIPO_PAGA is
            when ORARIA =>
                        ORE : FLOAT;
                        RETR_ORARIA : FLOAT;
            when STIPENDIO =>
                        RETR_MENSILE : FLOAT;
            when others => NULL;
    end record;
PRESIDENTE : IMPIEGATO(STIPENDIO);
AUTISTA : IMPIEGATO(ORARIA);
```

Volendo cambiare variante ad una variabile record è necessario assegnare un nuovo agglomerato comprensivo di discriminante.

Esempio:

```
PRESIDENTE : IMPIEGATO(ORARIA,40,70000.0);
```

Si osservi che ad una variabile di tipo record con varianti viene assegnato un blocco di memoria della dimensione della variante

massima, a meno che il compilatore non riesca a determinare che la variante non cambierà durante l'esecuzione del programma.

Puntatori

Il tipo di dato puntatore in ADA ha il nome di **access**. Una variabile non può essere dichiarata direttamente di un tipo puntatore, ma bisogna usare una definizione di tipo.

```
type <nome_del_tipo_puntatore> is access <nome_tipo>;
```

Un caso in cui torna utile il poter disporre di un tale tipo è quello in cui si vogliono gestire delle liste (code, pile)

Esempio:

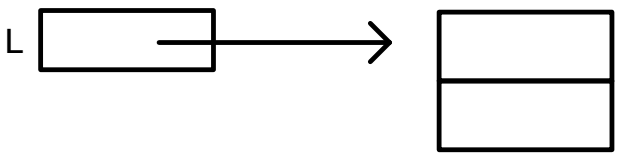
```
type CELLA;
type LINK is access CELLA;
type CELLA is
    record
        VALORE : INTEGER;
        PROSSIMA : LINK;
    end record;
L : LINK;                    (equivale a L : LINK := null;)
```

Da un punto di vista grafico la situazione a questo punto sarà:

Per creare un oggetto di tipo `CELLA` occorrerà eseguire la funzione allocatore **new**.

```
L := new CELLA;
```

La situazione a questo punto sarà:



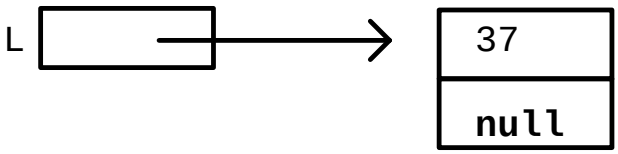
null

A questo punto, per assegnare un valore alla cella puntata da L, si dovrà scrivere:

```
L.VALORE := 37;
```

In alternativa potevamo passare dei “parametri” alla funzione allocatore:

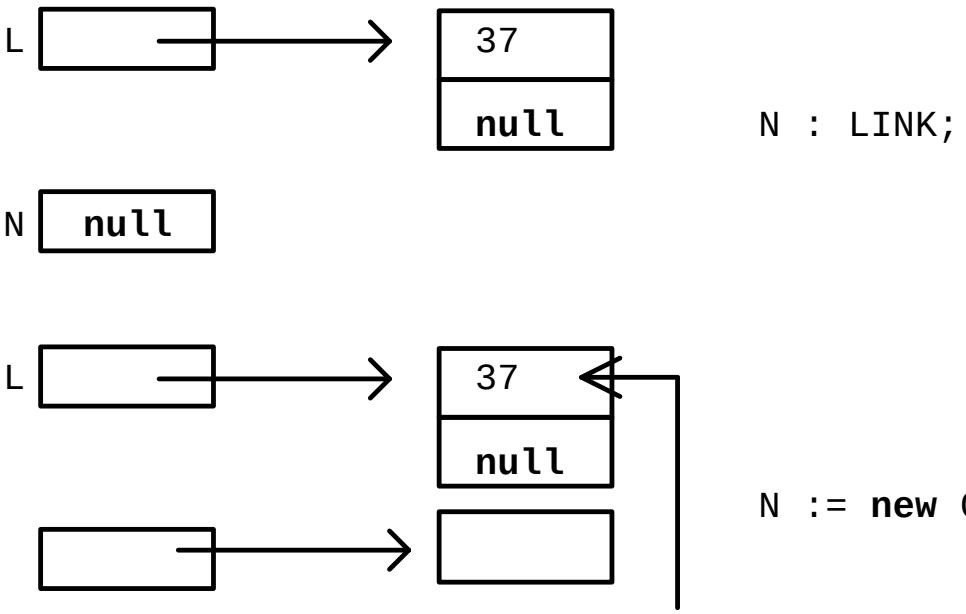
```
L := new CELLA’(37,null);
```

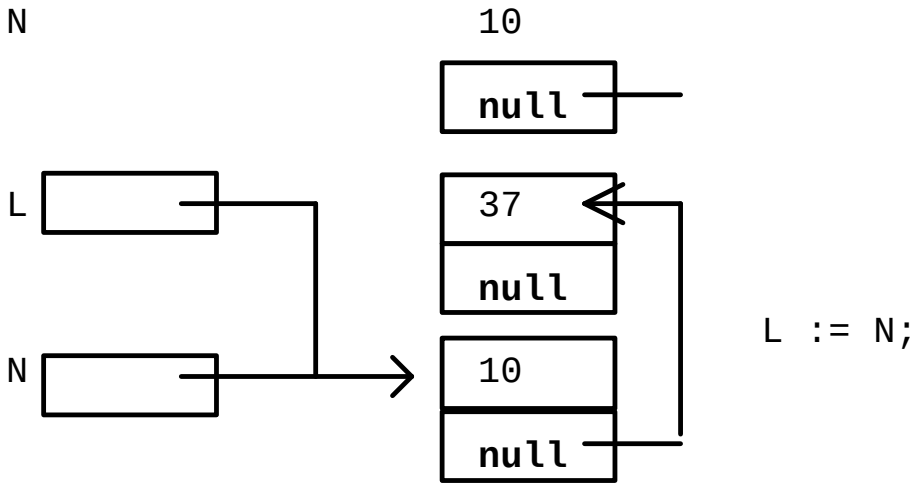


Creiamo adesso un nuovo record e concateniamolo a quello creato poco fa:

```
N : LINK;
N := new CELLA’(10,L);
L := N;
```

Graficamente:





Si osservi che quest'ultima istruzione copia i valori di accesso. Un'assegnamento dei valori del record puntato da `N` ai relativi campi del record puntato da `L` si sarebbe dovuto scrivere:

```
L.VALUE := N.VALUE;
L.NEXT := N.NEXT;
```

oppure

```
L.all := N.all;
```

Operatori per Espressioni

```
 **                                                          (exp)
 *    /    mod     rem
 +    -    abs     not                                    (+ e - unari)
 +    -     &                                             (+  e - binari)
 =    /=    <      <=        >     >=   in  not in
and   or   xor  and then  or else
```

Istruzioni Condizionali

```
if <bool_expr> then
    <lista_comandi>
elseif <bool_expr> then
    <lista_comandi>
elseif <bool_expr> then
    ......
else
    <lista_comandi>
end if;
```

Istruzioni Case

```
case <espressione> is
   when <valore>|<valore>|...|<valore> => <lista_comandi>
   when <valore>|<valore>|...|<valore> => <lista_comandi>
    ......
   when others => <lista_comandi>
end case;
```

Si osservi che il **case** ADA deve contemplare tutti i valori possibili che <espressione> può assumere.

Esempio:

```
type COLORI is (ROSSO,VERDE,BLU,NERO,BIANCO,GIALLO);
COLORE : COLORI;
case COLORE is
    when ROSSO => <lista_comandi>1
    when VERDE|NERO|GIALLO => <lista_comandi>2
    when others => null;
end case;
```

Istruzioni Iterative

Il costrutto iterativo di base è un loop infinito.

```
[<etichetta> :]  loop
                     <lista_comandi>
                 end loop;
```

Esempio:

```
loop
    LAVORA;
    MANGIA;
    DORME;
end loop;
```

Si può uscire da un loop a causa di un'eccezione, oppure eseguendo i comandi **`exit`**, **`goto`**, **`return`**.

La sintassi dell'istruzione **`exit`** è:

```
exit [[<etichetta>] when <bool_expr>];
```

Esempio:

```
VITA_MONOTONA : loop
                    LAVORA;
                    MANGIA;
                    DORME;
                    exit VITA_MONOTONA when basta();
                end loop;
```

Il costrutto **loop** può essere trasformato in un “while” o un “for” con una semplice modifica dell’intestazione. La sintassi del comando può essere riscritta nel modo seguente:

```
[<etichetta> :] [<intestazione>]  loop
                                      <lista_comandi>
                                  end loop;
< intestazione > ::=
        while <bool_expr> |
        for <variabile> in <intervallo_discreto> |
        for <variabile> in reverse <intervallo_discreto>
```

A differenza delle altre variabili la `<variabile>` che governa il ciclo del for non deve essere dichiarata. Essa viene creata all’inizio dell’esecuzione dell’istruzione e scompare alla fine del ciclo.

Esempi

1)
```
VITA_MONOTONA : while not basta() loop
                    LAVORA;
                    MANGIA;
                    DORME;
                end loop;
```

2)
```
for GIORNO in (LUN,MAR,MER,GIO,VEN,SAB,DOM) loop
    case GIORNO is
        when LUN..VEN =>
                    LAVORA;
                    MANGIA;
                    DORME;
        when others =>
                    OZIA;
  end loop;
```

Sottoprogrammi: Funzioni e Procedure

La sintassi per la definizione di funzioni e procedure è:

```
function <nome>(<parametri_formali>) return <tipo> is
    <dichiarazioni>;
begin
    <lista_comandi>;
exception
    <gestori_eccezioni>;
end;

procedure <nome>(<parametri_formali>) is
    <dichiarazioni>;
begin
    <lista_comandi>;
exception
    <gestori_eccezioni>;
end;

<parametri_formali> ::=
<parametro>: <metodo_trasm> <tipo> [,<parametri_formali>]

<metodo_trasm> ::= in | out | in out
```

Le funzioni possono avere solo parametri e possono restituire un valore del tipo specificato nell'intestazione.

Esempio

```
type VETTORE is array (INTEGER range <>) of REAL;
function SOMMA(A:VETTORE) return REAL is
    RISULTATO : REAL;
begin
    for I in A'RANGE loop
        RISULTATO := RISULTATO + A(I);
    end loop;
    return RISULTATO;
end SOMMA;
V:VETTORE(1..4):=(1.0,2.0,3.0,4.0);
```

```
S:REAL;
...
S := SUM(V);
```

La trasmissione dei parametri può avvenire in tre modi:

??***Corrispondenza Posizionale***

È il metodo tipico secondo il quale i valori attuali in una chiamata di una procedura/funzione vengono assegnati ai parametri formali seguendo l'ordine con cui sono stati dichiarati.

??***Corrispondenza Nominale***

Un valore attuale può essere preceduto da:

```
<nome_param> =>
```

In questo modo il parametro al quale andrà assegnato il valore è indicato esplicitamente per cui i valori attuali possono essere specificati in qualsiasi ordine.

??***Valori per Default***

Se nella dichiarazione di un parametro **`in`** viene specificato un valore di default, in sede di invocazione del sottoprogramma il valore attuale relativo a questo parametro può essere omesso.

```
<nome_param> : in <tipo> := <valore_default>
```

Package

Nei linguaggi di programmazione tradizionali il controllo della visibilità è molto limitato. Se il programma prevede l'accesso ad un sottoprogramma, si ha la possibilità di accedere anche alle variabili appartenenti a questo.

Esempio

```
MAX : constant := 100;
S : array (1..MAX) of INTEGER;
TOP : INTEGER range 0..MAX;

procedure PUSH (X:INTEGER) is
begin
    TOP := TOP + 1;
    S(TOP) := X;
end PUSH;

function POP return INTEGER is
begin
    TOP := TOP - 1;
    return S(TOP + 1);
end POP;
```

Nell'esempio possiamo modificare la pila attraverso l'uso di `PUSH` e `POP`, ma possiamo anche modificare direttamente `S` giacché ne conosciamo la struttura.

ADA fornisce un potente meccanismo per proteggere le dichiarazioni e risolvere questo problema: i ***packages***. Attraverso i packages è possibile indicare qual è la parte che si vuol rendere visibile (***parte dichiarativa***) e qual è invece quella privata (***corpo del package***); esiste infine una terza parte per l'***inizializzazione*** del package.

Esempio

```
package STACK is
    procedure PUSH (X:INTEGER);
    function POP return INTEGER;
end STACK;

package body STACK is
    MAX : constant := 100;
    S : array (1..MAX) of INTEGER;
    TOP : INTEGER range 0..MAX;

    procedure PUSH (X:INTEGER) is
    begin
        TOP := TOP + 1;
        S(TOP) := X;
    end PUSH;

    function POP return INTEGER is
    begin
        TOP := TOP - 1;
        return S(TOP + 1);
    end POP;

begin
    TOP := 0;
end STACK;
```

Il corpo di ogni sottoprogramma specificato nella parte specificativa dovrà essere scritto nel corpo del package; si possono però definire anche sottoprogrammi nel corpo senza una specifica nella parte dichiarativa: si tratta tipicamente di procedure e funzioni di servizio del relativo package.

Cosa vuol dire “eseguire un package”?
Vuol dire eseguire le dichiarazioni di tipi, variabili , costanti, sottoprogrammi contenuti nel suo corpo, cosicché la parte del programma che li ha richiesti li possa utilizzare.

È possibile dichiarare un package all’interno di un altro package:

1. Inserendo la parte specificativa del package P_2 all'interno di quella del package P_1 e il corpo del package P_2 all'interno del corpo del package P_1.
2. Inserendo la parte specificativa ed il corpo del package P_2 all'interno del corpo del package P_1.

NOTA BENE:
La specifica di un package può contenere soltanto specifiche e non corpi.

Per riferirsi agli elementi visibili di un package bisogna per forza nominare anche il package.

Esempio
```
STACK.PUSH(3);
Z := STACK.POP;
```

Se però si inserisce nella parte dichiarativa del blocco che usa il package la riga:

```
use STACK;
```

ci si può riferire direttamente agli elementi di `STACK`.

Esempio
```
PUSH(3);
Z := POP;
```

Questo vale a meno di ambiguità che il compilatore non è in grado di risolvere come ad esempio due pile `STACK1` e `STACK2` di interi, entrambe con le proprie `PUSH` e `POP`: se si omette il riferimento al package di appartenenza e si invoca la `POP`, come si deduce il package al quale ci si vuol riferire?

Nel caso in cui si vogliano dichiarare solamente dei tipi, delle costanti e delle variabili, non occorre specificare un corpo per il package.

Esempio

```
package GIORNI is
    type GIORNO is (LUN,MAR,MER,GIO,VEN,SAB,DOM);
    subtype FERIALE is range LUN..VEN;
    DOMANI : constant array (GIORNO) of GIORNO :=
             (MAR,MER,GIO,VEN,SAB,DOM,LUN);
    SUCC_FERIALE : constant array (FERIALE) of FERIALE :=
             (MAR,MER,GIO,VEN,LUN);
end GIORNI;
```

I packages sono molto utili in tutti quei casi in cui un programma si compone di più parti compilate in fasi distinte.

Library Units

Quando scriviamo un programma, solitamente facciamo affidamento sul fatto che esistano già parti di codice da poter riutilizzare. La collezione di questi programmi è detta ***libreria***.
In ADA ci sono le unità libreria, cioè le specifiche di packages o di sottoprogrammi, e le unità secondarie, cioè il corpo di detti packages e sottoprogrammi.
Queste unità (libreria e secondarie) possono essere compilate insieme o separate. Tendenzialmente si preferisce la compilazione separata.
Dopo essere state compilate le unità entrano a far parte delle librerie di programmi e potranno essere richiamate attraverso la clausola **`with`**.

Esempio:

```
with stack;
procedure main is
    use stack; is range LUN..VEN;
    m, n : integer;
begin
     ...fa qualcosa con la pila
end main;
```

Tutti i programmi sono implicitamente dipendenti dal package STANDARD, per cui la clausola "`with STANDARD;`" può essere omessa.

La clausola **`with`** deve precedere staticamente l'inizio dell'unità che la utilizzerà.

Una clausola **`with`** può far riferimento a più unità

```
with pippo, pluto;
```

oppure, equivalentemente, si possono concatenare più clausole **`with`** `pippo;`

```
with pluto;.
```

Con una clausola **`with`** si indicano solo le unità dalle quali si dipende direttamente; se poi queste dovessero dipendere da ulteriori unità, vorrà dire che al loro interno ci saranno le relative clausole **`with`**, ma la cosa è del tutto trasparente.

Esempio:

```
with pippo;
package pluto is....

with pluto;
package paperino is........
```

Infine la clausola che precede la specifica di un package o di un sottoprogramma si applica tanto alla specifica quanto al corpo. Tuttavia si può ripetere prima della dichiarazione del corpo o in

alternativa si possono aggiungere altre clausole (diverse cioè da quelle della specifica).

In caso di compilazione separata della specifica e del corpo di un package, la compilazione della specifica deve sempre precedere quella del corpo.

Inoltre la clausola **`with`** stabilisce una dipendenza con la specifica dell'unità e non con il suo corpo. Questo vuol dire che un'eventuale ricompilazione del corpo di un unità non richiede la ricompilazione della specifica, a meno di modifiche in quest'ultima. In caso invece di modifiche della specifica, bisognerà ricompilare tutte le unità da essa dipendenti.

Nel caso delle unità librerie non è consentito il sovraccarico (overloading) dei nomi.

Subunits

Uno dei casi in cui bisogna ricorrere alla compilazione separata è nei grandi progetti. Tipicamente in questi casi un programma non viene costruito in blocco, ma frazionato in progetti di entità più ragionevole, ognuno dei quali viene assegnato ad un gruppo che ne segue lo sviluppo del codice relativo e la successiva manutenzione.

Il meccanismo posseduto da ADA che rende possibile questa linea di progettazione è quello degli ***stub*** (moncherino, troncone) e consiste nel rimuovere il corpo di un package o di un sottoprogramma dall'unità libreria della quale fa parte per compilarlo separatamente.

Esempio

```
package body STACK is
    MAX : constant := 100;
    S : array (1..MAX) of INTEGER;
    TOP : INTEGER range 0..MAX;
    procedure PUSH (X:INTEGER) is separate;
    function POP return INTEGER is separate;
begin
    TOP := 0;
end STACK;
```

Le parti rimosse per essere sottoposte a compilazione separata prendono il nome di ***sottounità***. Esse verranno dichiarate da qualche altra parte e, per permettere al compilatore di risalire all'unità della quale facevano parte, saranno precedute dalla parola chiave separate seguita dal nome dell'unità.

Esempio

```
separate (STACK)
procedure PUSH (X:INTEGER) is
begin
    TOP := TOP + 1;
    S(TOP) := X;
end PUSH;

separate (STACK)
function POP return INTEGER is
begin
    TOP := TOP - 1;
    return S(TOP + 1);
end POP;
```

Nel caso in cui `STACK` fosse stata una sottounità di un'altra unità (sia essa `LISTE`), avremmo dovuto indicare questa appartenenza nelle clausole **`separate`** per `PUSH` e `POP`, scrivendo:

```
separate (LISTE.STACK)
```

In maniera analoga alle unità libreria, non si possono avere sottounità con lo stesso nome.
Le sottounità dipendono dalle unità alle quali appartengono, per cui devono sempre essere compilate dopo di dette unità.

La sottounità eredita eventuali dipendenze da altre unità da parte dell'unità alla quale appartiene. Ad esempio se `STACK` fosse dipesa da un package `VETTORI`, non sarebbe stato necessario ripetere la clausola **`with`** `VETTORI` prima della definizione di `PUSH` e prima di `POP`.
Nulla vieta ad una sottounità di avere dipendenze maggiori dell'unità alla quale appartiene: le relative clausole devono in questo caso precedere la **`separate`** `(STACK)`.

Esempio

```
with mio_package; use mio_package;
separate (STACK)
procedure PUSH (X:INTEGER) is
begin
    TOP := TOP + 1;
    S(TOP) := X;
end PUSH;

separate (STACK)
function POP return INTEGER is
begin
    TOP := TOP - 1;
    return S(TOP + 1);
end POP;
```

Renaming

Se l'overloading consente di gestire i casi di omonimia, esiste la possibilità anche di definire dei sinonimi.

Vediamo, ad esempio come definire dei sinonimi per evitare l'uso delle estensioni nell'invocazioni dei sottoprogrammi contenuti in un package senza ricorrere alla clausola **use**.

Esempio

```
declare
    procedure mia_push (i:integer) renames stack.push;
    function mia_pop return integer renames stack.pop;
begin
    ..
    mia_push(4);
    ..
    k := mia_pop;
end;
```

Si osservi che i parametri di `mia_push` e di `mia_pop` sono gli stessi dei sottoprogrammi che rinominano.

Il poter disporre di un meccanismo di rinominazione ci è utile anche in quei casi in cui le omonimie ci creano dei problemi (es. compilazione separata di sue sottoprogrammi con lo stesso nome).

Si può usare questo meccanismo anche per valutare parzialmente variabili e costanti.

Esempio

```
for i in persona'range loop
    put(persona(i).nato.giorno);put(":");
    put(nome_mese'val(persona(i).nato.mese - 1));
    put(":");put(persona(i).nato.anno);
end loop;

for i in persona'range loop
    declare
        d : data renames persona(i).nato;
    begin
        put(d.giorno);put(":");
        put(nome_mese'val(d.mese - 1));
        put(":");put(d.anno);
    end
```

```
end loop;
```

La ridenominazione si applica a variabili, costanti, package e sottoprogrammi, ma non ai tipi. Per quest'ultimi la ridenominazione può comunque essere ottenuta attraverso la definizione di un sottotipo:

```
subtype nome_sottotipo is nome_tipo;
```

Tipi Riservati

Il meccanismo dei packages è molto utile per non consentire al programmatore di vedere come un dato è fatto pur consentendogli di utilizzarlo.

Esempio

```
package numeri_complessi is
    type num_compl is
        record
            parte_reale, parte_imm: real;
        end record;
    i: constant num_compl := (0.0,1.0);
    function "+" (x,y:num_compl) return num_compl;
    function "-" (x,y:num_compl) return num_compl;
    ...
end numeri_complessi;
```

Il problema di una tale rappresentazione è che il programmatore, sapendo che i numeri complessi sono realizzati nella forma cartesiana, potrebbe eseguire operazioni su di essi non solo attraverso le funzioni "+" e "-" definite nel package, ma anche scrivendo:

```
x.parte_imm := x.parte_imm + 10;
```

invece di

```
x := x + <rappresentazione di 10i>;
```

Il problema è che se si decide di passare dalla rappresentazione cartesiana a quella polare non basterà semplicemente riscrivere il package `numeri_complessi`.

Si può allora nascondere al programmatore come è costruito il tipo dei complessi nel modo seguente:

Esempio

```
package numeri_complessi is
    type num_compl is private;
    i: constant num_compl;
    function "+" (x,y:num_compl) return num_compl;
    function "-" (x,y:num_compl) return num_compl;
    function "*" (x,y:num_compl) return num_compl;
    function "/" (x,y:num_compl) return num_compl;
    function crea_compl (r,i:real) return num_compl;
    function parte_reale (x:num_compl) return real;
    function parte_imm (x:num_compl) return real;
private
    type num_compl is
        record
            parte_reale, parte_imm: real;
        end record;
    i: constant num_compl := (0.0,1.0);
end numeri_complessi;
package body numeri_complessi is
    function "+" (x,y:num_compl) return num_compl is
    begin
        return (x.parte_reale + y.parte_reale,
                x.parte_imm + y.parte_imm);
    end "+";
    .......
end numeri_complessi;
```

L'aver dichiarato il tipo `num_compl` come private fa si che all'esterno del package `numeri_complessi` le uniche operazioni definite su un numero complesso sono l'assegnamento, "=", "/=" e le operazioni definite nella parte visibile del package.

Si osservi che la costante `i` è stata solo specificata nella parte visibile, mentre il suo valore è stato definito successivamente nella parte privata: infatti è solo a questo punto che si sa come è fatto il tipo di `i` e quindi come deve essere fatto il valore da assegnarle: si dice che `i` è una ***costante differita***.
A questo punto il programmatore potrebbe scrivere:

Esempio

```
declare
    use numeri_complessi;
    a_c,b_c: num_compl;
    a_r,b_r: real;
begin
    a_c := crea_compl(3.7,12.0);
    b_c := a_c + i;
    a_r := parte_reale (a_c) + 127.043;
    a_c := crea_compl(3.0,0.0) * a_c; (molt. per costante)
    .....
end;
```

A questo punto se si decidesse di cambiare rappresentazione dei complessi dalla cartesiana alla polare basterà cambiare solo la parte privata del package.

Esempio

```
private
    pi_greco : constant := 3.14159
    type num_compl is
        record
            x : real;
            teta : real range 0.0..2.0 * pi_greco;
        end record;
    i: constant num_compl := (1.0,0.5 * pi_greco);
end numeri_complessi;
```

A questo punto bisognerà riscrivere anche il corpo del package `numeri_complessi`.

Si osservi che il programma di un utente che usa questo package non dovrà essere modificato, sebbene dovrà essere ricompilato.

Un tipo privato viene definito in due passi:
??la specifica,
??la dichiarazione.
Nel periodo che intercorre tra questi due momenti si può nominare il tipo per definire costanti (senza assegnazione di valore), parametri di sottoprogrammi, funzioni, ma non per definire delle variabili. A quest'epoca infatti non si sa ancora come è realmente fatto il tipo e quindi non ha senso assegnare valori di questo tipo.

Si possono però specificare nella lista dei parametri dei sottoprogrammi delle espressioni che identificano valori di default per oggetti di un tipo privato. Questo è possibile in quanto la valutazione di queste espressioni verrà fatta soltanto quando il sottoprogramma verrà invocato.

Tipi Riservati Limitati

Le uniche operazioni possibili su un tipo riservato sono (a parte quelle definite dai sottoprogrammi specificati nella parte visibile) l'assegnamento, "=", "/=".

Si può restringere ulteriormente questo insieme introducendo le cosiddette limitazioni.
Eventuali operatori "=" e "/=" possono essere inseriti nel package attraverso la definizione delle rispettive funzioni a valori booleani.

Per quanto riguarda l'assegnamento, invece, può essere gestito dal package attraverso le modalità dei parametri dei sottoprogrammi (gli effetti collaterali sulle variabili non sono considerati degli assegnamenti).

Esempio

```
procedure mio_ass
        (var:in out num_compl, cost:in num_compl) is
    var := var + cost;
end mio_ass;
```

Esempio

```
package pila is
    type pila is limited private;
    procedure push (s : in out pila, x : in integer);
    procedure pop (s : in out pila, x : out integer);
    function "=" (s, t : pila) return boolean;
private
    max : constant := 100;
    type vettore_di_interi is
        array (integer range <>) of integer;
    type pila is
        record
            s : vettore_di_interi(1..max);
            testa : integer range 1..max := 0;
        end record;
end pila;
package body pila is
    procedure push (s : in out pila, x : in integer) is
    begin
        s.testa := s.testa + 1;
        s.s(s.testa) := x;
    end push;
    procedure pop (s : in out pila, x : out integer) is
    begin
        x := s.s(s.testa);
        s.testa := s.testa - 1;
    end pop;
    function "=" (s, t : pila) return boolean is
    begin
        if s.testa /= t.testa then
            return false;
        end if;
        for i in 1..s.testa loop
            if s.s(i) /= t.s(i) then
                return false;
            end if;
```

```
            end loop;
            return true;
        end "=";
    end pila;
```

... esempio di uso ...

```
declare
    use pila;
    p : pila;
    vuota : pila;
begin
    push (p,300);
    .....
    pop(p,var_intera);
    ...
    if p = vuota then
        ...
    end if;
    ...
end;
```

N.B.: `vuota` è la pila vuota che è stata dichiarata sotto forma di variabile. Sarebbe stata più indicata una costante, ma non possiamo definire costanti di un tipo privato fuori dal package al quale il tipo appartiene. Bisogna:

??definire la costante nel package,

??definire una funzione a valori booleani `"vuota"` per testare se una pila è vuota.

Esempio

Vediamo adesso un esempio che usa i tipi riservati e limitati per gestire il *problema delle n-risorse* (tralasciando la concorrenza).

```
package gestore is
    type risorsa is limited private;
    procedure acquisizione (r : in out risorsa);
    procedure rilascio (r : in out risorsa);
    procedure uso (r : in out risorsa; .... );
    function risorsa_valida (r : risorsa) return boolean;
    .....
private
    max : constant := 100;
    subtype id_risorsa is integer range 1..max;
    type risorsa is
        record
            id : id_risorsa := 0;
        end record;
```

```
end gestore;
package body gestore is
    risorse_libere : array
            (id_risorsa range 1..id_risorsa'last)
                of boolean (others => true);
    function risorsa_valida (r : risorsa)
                        return boolean is
    begin
        return r.id /= 0;
    end risorsa_valida;
    procedure acquisizione (r : in out risorsa) is
    begin
        if r.id = 0 then
            for i in risorse_libere'range loop
                if risorse_libere(i) then
                    risorse_libere(i):=false;
                    r.id := i;
                    return;
                end if;
            end loop;
        end if;
    end acquisizione;
    procedure rilascio (r : in out risorsa) is
    begin
        if r.id /= 0 then
            risorsa_libera(r.id) := true;
            r.id := 0;
        end if;
    end rilascio;
    .....
    procedure uso (r  : in out risorsa; ....) is
    begin
        if risorsa_valida(r) then
            <uso risorsa>
        end if;
    end uso;
end gestore;
```

Il protocollo per l'uso di una risorsa sarà:

```
declare
    use gestore;
    mia_risorsa : risorsa;
begin
    acquisizione (mia_risorsa);
```

```
    uso (mia_risorsa; ....);
    rilascio (mia_risorsa);
end;
```

Si osservi che il tipo risorsa è limitato, per cui l'utente non ha idea di quale risorsa gli sia stata assegnata. L'unica cosa di cui dispone è la funzione `risorsa_valida` attraverso la quale può verificare se una risorsa gli è stata assegnata.

Le Eccezioni

Di fronte ad una violazione delle regole del linguaggio viene segnalata una eccezione di:

CONSTRAINT_ERROR

corrisponde ad una violazione degli estremi di un intervallo;

NUMERIC_ERROR

corrisponde ad un errore in una operazione aritmetica (es. div. per 0);

PROGRAM_ERROR

corrisponde ad una violazione della struttura di controllo delle operazioni (es. chiamata ad una procedura la cui dichiarazione non è stata ancora eseguita);

STORAGE_ERROR

corrisponde ad un supero dei limiti di memoria previsti;

TASKING_ERROR

corrisponde ad un errore derivante dalla gestione dei tasks.,

Attraverso un ***gestore delle eccezioni*** riusciamo a rimediare a queste situazioni di errore; la sua struttura sintattica è:

```
    begin
        <istruzioni>
    exception
        when <eccezione> => <istruzioni>;
        ...
        when <eccezione> => <istruzioni>;
```

```
        when others => <istruzioni>;
    end;
```

"Brutto Esempio" (il package `GIORNI` è a pag. 31)

```
with GIORNI; use GIORNI;
begin
   domani := giorno'succ(oggi);
exception
   when CONSTRAINT_ERROR => domani := giorno'first;
end;
```

I gestori delle eccezioni possono essere definiti al termine di ogni blocco, sottoprogramma, package, ecc. e stabiliscono come terminare l'esecuzione della sezione nella quale si trovavano in caso di eccezione.

Quello che deve essere chiaro è che l'eccezione fa terminare l'esecuzione della sezione nella quale si verifica:

- se in questa sezione è presente un gestore per l'eccezione verificatasi, questo eseguirà delle operazioni alternative e poi ritornerà il controllo alla sezione che aveva attivato questa sezione;
- se invece non è presente alcun gestore l'esecuzione della sezione viene comunque terminata e l'eccezione viene passata alla sezione invocante. Se questa contiene un gestore, l'eccezione sarà gestita a questo livello, altrimenti si passa al livello superiore e così via.
- se si arriva all'ultimo livello senza incontrare un gestore, si ha la terminazione del programma principale ed il sistema operativo (sempre che ve ne sia uno!) restituirà un messaggio di errore.

Se si verifica un'eccezione all'interno di un gestore, essa verrà trattata dal gestore del livello superiore.

Dichiarazione e Generazione di Eccezioni

Introduciamo l'argomento con un esempio. Scriviamo un blocco che utilizzi il package STACK di pag. 29.

```
declare
    use STACK;
begin
    ....
    push (m);
    ....
    n:=pop;
    ....
exception
    when CONSTRAINT_ERROR => Sarà stata la pila???
end;
```

STACK non contiene nessun gestore delle eccezioni; inoltre STACK non contiene nessun controllo che prevenga il superamento dei limiti di dimensioni della pila e se si verificasse un tale evento, verrebbe generata una eccezione di CONSTRAINT_ERROR, la quale verrebbe ritornata al blocco qui sopra che ovviamente non saprebbe stabilire chi ha causato una tale eccezione.

In ADA, oltre alle classi di eccezioni predefinite è possibile definirne delle altre e generarle da programma.

La definizione di una nuova eccezione deve essere fatta nella parte dichiarativa di un blocco, di un sottoprogramma, di un package e via dicendo e la sintassi è:

```
    nome_eccezione : exception;
```

Per attivare il gestore delle eccezioni su una data eccezione, si può usare l'istruzione:

```
    raise nome_eccezione;
```

Esempio

```
package STACK is
    STACK_ERROR : exception;
    procedure PUSH (X:INTEGER);
    function POP return INTEGER;
end STACK;

package body STACK is
    MAX : constant := 100;
    S : array (1..MAX) of INTEGER;
    TOP : INTEGER range 0..MAX;

    procedure PUSH (X:INTEGER) is
    begin
        if TOP = MAX then
            raise STACK_ERROR;
        end if;
        TOP := TOP + 1;
        S(TOP) := X;
    end PUSH;

    function POP return INTEGER is
    begin
        if TOP = 0 then
            raise STACK_ERROR;
        end if;
        TOP := TOP - 1;
        return S(TOP + 1);
    end POP;

begin
    TOP := 0;
end STACK;
```

*** blocco con gestore ***

```
declare
use STACK;
```

✍ se manca, l'eccezione è `STACK.STACK_ERROR`

```
begin
    ....
    push (m);
    ....
    n:=pop;
    ....
exception
```

```
    when STACK_ERROR => errore sulla pila
    when others => altri errori
end;
```

Un'istruzione **`raise`** può comparire all'interno di un gestore e rappresenta (informalmente) una chiamata al gestore del livello superiore. In questo caso, però, deve comparire direttamente e non può comparire in eventuali sottoprogrammi richiamati dal gestore.

Inoltre se **`raise`** non specifica nessuna eccezione l'effetto è quello di far filtrare l'eccezione anche al livello superiore: si può quindi gestire un'eccezione a più livelli.

È possibile rinominare un'eccezione; ad esempio nel caso della `STACK_ERROR` potremmo scrivere:

```
    PERICOLO : exception renames STACK.STACK_ERROR
```

Attenzione: si ricordi che il meccanismo delle eccezioni è pesante, per cui se ne raccomanda un uso moderato.

I Tasks

Un ***task*** descrive un'attività che può essere realizzata in concorrenza (in parallelo) con altre.
Sintatticamente un task è simile ad un package: anch'esso infatti si compone di una parte dichiarativa e di un corpo, solo che in questo caso la prima descrive l'interazione con gli altri tasks, mentre il corpo descrive il comportamento del task:

```
task <nome_task> [is
    ....
end <nome_task>];

task body <nome_task> is
    ....
end <nome_task>;
```

Come si vede dalla sintassi data per la specifica l'interazione con altri tasks è opzionale, nel qual caso la specifica di un task `T` sarà semplicemente:

```
    task T;
```

Esempio

Esecuzione di tre attività parallele senza interazione.

```
procedure attività_parallele is
    task attività_1;
    task body attività_1 is
        ....
    end attività_1;

    task attività_2;
    task body attività_2 is
        ....
    end attività_2;

begin
    attività_3;
end;
```

Quando `attività_parallele` va in esecuzione, i tasks `attività_1` e `attività_2` vengono definiti e la loro esecuzione avrà inizio solo quando viene eseguito il **`begin`** del corpo della procedura: l'attività parallela ha inizio quindi con l'esecuzione del **`begin`**.

L'attività parallela invece ha termine quando tutti i tasks eseguono l'**`end`**, con conseguente distruzione di tutti i tasks. Il tasks principale (nell'esempio quello che esegue l'`attività_3`) attende sempre la terminazione dei task dipendenti (nell'esempio `attività_1` e `attività_2`).

Il Rendezvous

L'interazione in ADA è attuata attraverso il meccanismo del rendezvous: un task richiama una ***entry*** (unità di informazione) di un altro task.

Una entry è definita nella specifica di un task:

```
task T is
    entry E (<lista_parametri>)
end T;
```

dove i parametri possono essere "**`in`**", "**`out`**", "**`in out`**".

Una entry assomiglia ad una procedura e come una procedura può essere richiamata scrivendo:

```
T.E (<lista_valori>);
```

Per richiamare una entry bisogna sempre usare la notazione composta, poiché non è possibile usare la clausola **`end`** con un nome di task.

L'azione corrispondente ad una entry è definita nel corpo del task da una istruzione ***accept***:

```
accept E(<lista_parametri>) do
    ...
end E;
```

La struttura dell'accept con la presenza di una lista di parametri ci potrebbe far pensare ad una procedura. In realtà l'accept è una istruzione e non prevede, a meno di inserimento di blocchi, la possibilità di dichiarare variabili o costanti o di inserire una gestione delle eccezioni.

Anche l'esecuzione di una entry da parte di un task potrebbe far pensare alla chiamata di procedura. In realtà se un task chiama una procedura è lui stesso ad eseguire il codice relativo a quella procedura, mentre se esegue una entry, sarà il task proprietario di quella entry ad eseguire il relativo codice.

Interazione tra due task T_1 ***e*** T_2**.**

T_1 ✍ T_2.E(..)

T_2 ✍

```
accept E(..) do
    ...
end E;
```

Quando i due task eseguono queste istruzioni, si sincronizzano: il primo che arriva si sospende ad aspettare l'esecuzione dell'istruzione complementare da parte dell'altro.

Una volta sincronizzati, T_1 si sospende fino a quando T_2 non termina l'esecuzione dell'accept eseguendo la **end** dell'accept.

A questo punto riprende l'esecuzione parallela dei due tasks.

Esempio

```
task buffer is
    entry put (x : in elemento);
    entry get (x : out elemento);
end buffer;
task body buffer is
    y : elemento;
begin
    loop
        accept put (x : in elemento) do
            y := x;
        end put;
        accept get (x : out elemento) do
            x := y;
        end put;
    end loop;
end buffer;
```

Una entry può essere richiamata da più tasks contemporaneamente, ma l'esecuzione di una accept è relativa ad una sola entry. Questo comporta la sospensione dei tasks su una coda (=lista FIFO) in attesa dell'esecuzione della relativa accept.

Il task proprietario può sapere il numero di processi sospesi sulla coda della entry `E` testando l'attributo `COUNT` di `E`.

Ogni volta che il task proprietario della entry esegue una accept sblocca un task dalla coda della entry (se non ce ne sono si sospende in attesa di uno). Si osservi che lui non conosce il processo che va a sbloccare (***naming asimmetrico***).

Se una entry viene rinominata diventa una procedura; ad esempio

```
procedure put (x : in elemento) renames BUFFER.PUT;
```

Inoltre entry e accept possono essere dichiarate senza parametri; inoltre la accept può non avere una lista di istruzioni.

Esempio

```
task sync is
    entry signal;
end sync;
task body sync is
begin
    accept sync;
end sync;
```

Infine si osservi che se nel corpo di un'**accept** viene eseguita una **return**, si ottiene lo stesso effetto che si avrebbe con l'esecuzione della **end** dell'**accept**.

Il Task Scheduling

Normalmente un programma articolato su più tasks non dispone di un processore per ogni task. Bisogna allora prevedere un qualche meccanismo che consenta ai vari tasks di condividere la risorsa processore.

Esistono varie politiche di allocazione del processore. La più semplice è l'***allocazione per quanti di tempo***, che consiste nell'assegnare il processore a turno tra i vari tasks per un certo intervallo di tempo (es: 1 ms).

È necessario prevedere delle eccezioni alla regola: se un processo è in attesa su un rendez-vous, non ha senso assegnargli il processore, bisogna evitare, cioè, le ***attese attive***.

Questa politica di allocazione è troppo semplice, perché considera tutte le attività svolte dai vari tasks sullo stesso piano, cosa ben lontana dalla realtà. Occorre allora eseguire i tasks in base alla loro importanza e per fare questo si assegna ad ogni processo una priorità e il processore viene guadagnato dal processo con maggiore ***priorità*** (***politica di allocazione in base alla priorità***).

Prevedere solo il meccanismo delle priorità sarebbe ancora una volta troppo rigido: un task potrebbe guadagnare il processore e non consentirne più l'uso agli altri. Ecco perché di solito queste due politiche vengono combinate: ad esempio più processi possono avere la stessa priorità e tra questi il processore viene assegnato per quanti di tempo.

ADA adotta di base la politica di allocazione per quanti di tempo, mentre la politica per priorità è presente o meno a seconda della particolare implementazione del compilatore.

Quando un compilatore prevede la possibiltà di assegnare una priorità, la cosa può essere fatta attraverso un **pragma**. Nel caso dei tasks esso andrà inserito nella parte specificativa, mentre nel caso di un programma principale (è un task) nella parte dichiarativa più esterna.

Esempi

```
task T is
    pragma PRIORITY(10);
    entry E1 (...);
    ....
    entry En (...);
end T;

procedure P (...) is
    pragma PRIORITY(10);
     PRIORITY(2);
    ....
begin
    ....
end P;
```

L'espressione `PRIORITY` è statica: la priorità una volta assegnata non può più essere modificata. L'indice della priorità appartiene al sottotipo `PRIORITY` degli interi, ma l'intervallo di variazione di `PRIORITY` dipende dalla particolare implementazione; in tutti i casi, però, la priorità cresce con il crescere dell'indice.

La regola per lo scheduling dei tasks è la seguente (dal manuale di riferimento del linguaggio redatto dall'ANSI):
"*Se due tasks con diverse priorità sono entrambi eseguibili e potrebbero essere razionalmente eseguiti utilizzando gli stessi processori fisici e le stesse risorse elaborative, non può succedere che mentre è in corso l'esecuzione del task avente priorità più bassa non lo sia anche quello con priorità maggiore.*"

Si osservi che si parla solo di tasks con priorità diverse e che non si fa nessun riferimento ai casi di tasks aventi stessa priorità, nel qual caso l'implementatore è libero di scegliere il comportamento che ritiene più appropriato.

N.B.: la priorità non deve essere utilizzata per sincronizzare più tasks facendo discorsi del tipo "siccome il task A ha priorità 100 ed il task B priorità 2 posso concludere che A è sempre in esecuzione mentre B no" perché trascuro ad esempio la possibiltà che esistano più processori; per la sincronizzazione si usa il rendez-vous.

Scelta tra più Rendez-Vous

La **`accept`** va bene per problemi di sincronizzazione, ma non va bene per risolvere problemi in cui un task può dover scegliere tra impegnarsi in un rendez-vous o eseguire altre operazioni, oppure se un task può tovarsi a scegliere tra più rendez-vous.

Per casi come quelli sopra, ADA dispone dell'istruzione **`select`**. Questa istruzione si presenta in varie forme che permettono di realizzare meccanismi di interazione tra i processi molto potenti.

Come primo caso di **`select`**, consideriamo quello in cui si vule che un task non segua sempre lo stesso ordine nell'accettazione dei rendez-vous, come succedeva nel caso dell'esempio di pag. 52, in cui il task buffer accettava elementi sempre nello stesso ordine: prima una scrittura e poi una lettura.

Esempio

```
task variabile_condivisa is
    entry leggi (x : out elemento);
    entry scrivi (x : in elemento);
end variabile_condivisa;
```

```
task body variabile_condivisa is
    var : elemento;
begin
    accept scrivi (x : in elemento) do
        v := x;
    end;
    loop
        select
            accept leggi (x : out elemento) do
                x := v;
            end;
        or
            accept scrivi (x : in elemento) do
                v := x;
            end;
        end select;
    end loop;
end variabile_condivisa;
```

Il funzionamento della **`select`** è il seguente:

- se le code dei processi sospesi su `leggi` e di quelli sospesi su `scrivi` sono vuote, il task `variabile_condivisa` si sospende;
- se ci sono processi sospesi su `leggi`, ma non su `scrivi`, il task `variabile_condivisa` si impegnerà in un rendez-vous su `leggi` e viceversa;
- se ci sono sia processi sospesi su `leggi`, sia processi sospesi su `scrivi` ne viene scelto uno a caso (***non determinismo***)

Riguardo la frase “ne viene scelto uno a caso” bisogna chiarire che il meccanismo di scelta è a carico dell’implementatore del particolare compilatore e che deve essere teso ad evitare che i processi rimangano indefinitamente sospesi sulle code, si vuole cioè evitare la ***starvation***.

Un’altra forma di **`select`** prevede che il task, prima di impegnarsi in un rendez-vous, valuti se una certa condizione è soddisfatta.

Per illustrare questo caso facciamo l'esempio del buffer circolare (bounded buffer), in cui più tasks interagiscono su un array comune sul quale possono operare delle scritture e delle letture. Il buffer ha una lunghezza limitata, supponiamo di 10 posizioni, e viene riempito-vuotato circolarmente. Nell'inserire e rimuovere elementi dal buffer bisogna cercare di non superare i limiti e seguire una politica FIFO.

Esempio

```
task buffer_circolare is
    entry prelievo (x : out elemento);
    entry deposito (x : in elemento);
end buffer_circolare;

task body buffer_circolare is
    buffer : array (1..10) of elemento;
    leggi_da, scrivi_su: INTEGER range 1..10 := 1;
    num_elementi : INTEGER range 0..10 := 0;
begin
    loop
        select
            when num_elementi > 0 =>
            accept prelievo (x : out elemento) do
                x := buffer(leggi_da);
            end;
            leggi_da := (leggi_da mod 10) + 1;
            num_elementi := num_elementi - 1;
        or
            when num_elementi < 10 =>
            accept deposito (x : in elemento) do
                buffer(scrivi_su) := x;
            end;
            scrivi_su := (scrivi_su mod 10) + 1;
            num_elementi := num_elementi + 1;
        end select;
    end loop;
end buffer_circolare;
```

Le guardie della select hanno la forma:

```
            when condizione =>
                accept ....
```

```
end;
lista_istruzioni
```

Quando si incontra una **select** con guardie, vengono prima di tutto valutate le guardie (l'ordine di valutazione non è specificato) e solo i rami le cui guardie sono vere concorreranno nella scelta del rendez-vous; a questo punto il comportamento sarà lo stesso della **select** vista prima.

Si può vedere l'operazione di valutazione delle guardie come una riscrittura della **select** con guardie in una senza guardie e contenente solo i rami la cui guardia era vera.

Un ramo senza guardia è un ramo con guardia sempre vera.

Infine se tutte le guardie sono false viene generata un'eccezione PROGRAM_ERROR.

Si osservi che la valutazione delle guardie e l'accettazione del rendez-vous sono due fasi distinte e che quindi la condizione di una guardia potrebbe cambiare subito dopo la sua valutazione.

Per quanto riguarda la lista_istruzioni che segue la **accept** c'è da dire che questa va vista come una serie di operazioni da eseguire dopo aver eseguito il rendez-vous e prima di eseguirne un altro (è per questo che sono inserite nella **select**).

Esempio (Problema dei lettori-scrittori)

Si tratta di una variabile condivisa tra più processi; le operazioni di lettura possono avvenire in parallelo, mentre l'accesso è ristretto ad un solo processo in fase di scrittura.

```
package lettori_scrittori is
    procedure lettura (x : out elemento);
    procedure scrittura (x : in elemento);
```

```
end lettori_scrittori;

package body lettori_scrittori is
    variabile : elemento;
    task accesso is
        entry inizio_lettura;
        entry fine_lettura;
        entry scrittura (x : in elemento);
    end accesso;
    task body accesso is
        num_lettori : integer := 0;
    begin
        accept scrittura (x : in elemento) do
        variabile := x;
        end;
        loop
            select
                accept inizio_lettura;
                num_lettori := num_lettori + 1;
            or
                accept fine_lettura;
                num_lettori := num_lettori - 1;
            or
                when num_lettori = 0 =>
                accept scrittura (x : in elemento) do
                    variabile := x;
                end;
            end select;
        end loop;
    end accesso;

    procedure lettura (x : out elemento) is
    begin
        accesso.inizio_lettura;
        x := variabile;
        accesso.fine_lettura;
    end lettura;

    procedure scrittura (x : in elemento) is
    begin
        accesso.scrittura(x);
    end scrittura;

end lettori_scrittori;
```

La soluzione proposta al problema dei lettori-scrittori espone al rischio di attesa indefinita i processi di scrittura e, normalmente, una scrittura è un'operazione importante in un processo di elaborazione.

Si potrebbe apportare una modifica alla soluzione appena vista, bloccando le operazioni di lettura quando ci sono in coda processi di scrittura; basterà dunque riscrivere la **`select`** nel modo seguente:

```
select
    when scrittura'COUNT = 0 =>
    accept inizio_lettura;
    num_lettori := num_lettori + 1;
or
    accept fine_lettura;
    num_lettori := num_lettori - 1;
or
    when num_lettori = 0 =>
    accept scrittura (x : in elemento) do
        variabile := x;
    end;
end select;
```

L'attributo `COUNT` indica il numero di processi in attesa di un determinato rendez-vous (nel nostro caso di una scrittura). Questo attributo è da usare con cutela nelle guardie perché durante l'intervallo di tempo che intercorre tra la valutazione della guardia e l'accettazioe del rendez-vous, si possono aggiungere o rimuovere processi dalla coda di attesa, rischiando di portare il sistema allo stallo; nell'esempio sopra, comunque, l'attributo ha un comportamento sufficientemente corretto.

Select con Time-Out

Consente di stabilire un periodo di tempo entro il quale devono arrivare richieste di rendez-vous; se allo scadere di questo tempo non ci sono rendez-vous vengono eseguite delle istruzioni alternative.

```
select
    accept E1 ...;
or
    ....
or
    accept En ...;
or
    delay <tempo_in_ms>;
    <lista_istruzioni>
end select;
```

Nella stessa **`select`** possono esserci anche più rami **`delay`** ed in questo caso verrà scelto quello con intervallo minore.

Un ramo **`delay`** può essere protetto da una guardia.

L'inizio di un rendez-vous (non la conclusione) cancella il conteggio del time-out.

Si osservi che esiste anche un'istruzione **`delay`** il cui effetto è quello di provocare la sospensione di un task.

Select con Ramo Alternativo

Consente di indicare una sequenza di istruzioni da eseguire quando non è possibile scegliere nessuno degli altri rami della **select**. L'alternativa è sempre l'ultimo ramo di una **select** ed è unica. Inoltre non può essere abbinata a time-out.

```
select
    accept E1 ...;
or
    ....
or
    accept En ...;
else
    <lista_istruzioni>
end select;
```

Si osservi che mentre una **accept** (o una **delay**) in un ramo della **select** fa parte del meccanismo della **select** stessa, una **accept** (**delay**) in un ramo **else** viene eseguita incondizionatamente.

Esempio

```
select
    accept E1 ...;
or
    ....
or
    accept En ...;
or
    delay <tempo_in_ms>;
    <lista_istruzioni>
end select;
```

```
select
    accept E1 ...;
or
    ....
or
    accept En ...;
else
    delay <tempo_in_ms>;
    <lista_istruzioni>
end select;
```

Una **select** con alternativa non potrà mai generare un'eccezione di PROGRAM_ERROR quando tutte le guardie sono false, perché il ramo **else** è sempre vero.

Select per il Richiamo delle Entries

È possibile usare la **select** per richiamare una singola **entry** con time-out.

```
select
    E(...);
or
    delay <tempo_in_ms>;
    <lista_istruzioni>
end select;
```

Il task che esegue questa select attenderà per l'intervallo di tempo l'accettazione del rendez-vous da parte del task partner; trascorso l'intervallo senza successo il task richiedente eseguirà delle istruzioni alternative e riprenderà la sua esecuzione.

Esiste anche una **select** con alternativa per il richiamo di una singola **entry**:

```
select
    E(...);
else
    <lista_istruzioni>
end select;
```

Alcune precisazioni:

- una **select** per il richiamo di una **entry** si compone di due rami, uno contenente il richiamo della **entry** e l'altro contenente il time-out o l'alternativa;
- non è possibile inserire delle guardie nella **select** per il richiamo di una **entry**;
- non è possibile usare questo meccanismo per richiamare procedure e ***entries ridenominate come procedure***.

Bisogna prestare attenzione quando si abbina questo tipo di richiamo di entries con l'uso di attributi `COUNT` nelle guardie delle relative select del partner di rendez-vous. Una decisione presa su questo attributo può essere immediatamente contraddetta dall'uscita di un task dalla coda di attesa.

Esempio

Consideriamo la versione dei lettori_scrittori con la **`select`** di pag. 61 e riscriviamo la procedura di scrittura nel modo seguente:

```
procedure scrittura (x: in elemento, ok: out boolean) is
begin
    select
        accesso.scrittura(x);
        ok := true;
    or
        delay 10.0;
        ok := false;
    end select;
end scrittura;
```

Supponiamo ora che il task `T` sia l'unico task a poter operare delle scritture e che richieda una scrittura mentre è in corso una serie di letture: `T` si sospenderà in attesa della terminazione delle letture. Una volta terminate le letture, al ciclo successivo di esecuzione della select vengono valutate le guardie:

```
scrittura'COUNT = 0 ? false
```

poiché c'è un processo in attesa e

```
num_lettori = 0 ? true
```

in quanto è appena terminata una sequenza di letture.

Viene scelto di servire una lettura, ma prima che `accesso` si impegni nel rendez-vous, scade il time-out per `T` che lascia così la coda di attesa per la scrittura.

La situazione che si viene a creare è che `accesso` si blocca in attesa di una scrittura e blocca di conseguenza tutti i processi che

attendono di poter eseguire una lettura. Se a questo si aggiungesse una terminazione del task `T` ....
Riscriviamo il **`package`** `lettori_scrittori` sempre prevedendo il blocco delle letture in presenza di una sequenza di scrittura, ma senza l'uso dell'attributo `COUNT`. Si inserisce invece una entry comune che provvede a sospendere eventualmente i processi su una coda secondaria:

```
package body lettori_scrittori is
    variabile : elemento;
    type operazione is (lettura,scrittura);

    task accesso is
        entry inizio(op : operazione);
        entry fine_lettura;
        entry scrittura;
        entry fine_scrittura
    end accesso;

    task body accesso is
        num_scrittori : integer := 0;
        num_lettori : integer := 0;
    begin
        loop
            select
            when num_scrittori = 0 =>
                accept inizio(op : operazione) do
                    case op is
                        when lettura =>
                        num_lettori := num_lettori + 1;
                        when scrittura =>
                        num_scrittori := 1;
                    end case;
                end inizio;
            or
                accept fine_lettura;
                num_lettori := num_lettori - 1;
            or
                when num_lettori = 0 =>
                accept scrittura;
            or
                accept fine_scrittura;
```

```
                num_scrittori := 0;
            end select;
        end loop;
    end accesso;

    procedure lettura (x : out elemento) is
    begin
        accesso.inizio(lettura);
        x := variabile;
        accesso.fine_lettura;
    end lettura;

    procedure scrittura (x : in elemento) is
    begin
        accesso.inizio(scrittura);
        accesso.scrittura;
        variabile := x;
        accesso.fine_scrittura;
    end scrittura;

end lettori_scrittori;
```

Esercizi

1. Aggiungere l'inizializzazione della variabile condivisa facendo si che il primo rendez_vous accettato da `accesso` sia con un processo scrittore.

2. Riscrivere il package `lettori_scrittori` in modo che il task `accesso` abbia due sole entry:

```
task accesso is
    entry inizio(op : operazione);
    entry fine;
end accesso;
```

Tipi Tasks

Ci sono casi in cui è necessario creare un imprecisato numero di task diversi dal punto di vista di entità attive, ma con la stessa struttura sintattica. In questi casi si dovrebbe scrivere sempre lo stesso codice, cambiando soltanto il nome del task.

ADA offre la possibilità di definire un tipo task e di creare dei tasks in maniera analoga alla definizione delle variabili.

```
task type T is
    ....
end T;

task body T is
    ....
end T;
```

Per creare un task basato su questo tipo basterà scrivere:

```
mio_task : T;
```

Possiamo definire anche oggetti più complessi:

```
array_di_tasks : array (1..13) of T;
type record_con_task is
    record
        campo_task : T;
        ....
    end record;
var_rec_con_task : record_con_task;
type task_accesso is access T;
```

Gli oggetti task sono assimilabili alle costanti piuttosto che alle variabili: una volta dichiarato un oggetto di un tipo task questo non potrà più essere modificato. Inoltre l'assenza delle operazioni di assegnamento, di verifica dell'uguaglianza o della diversità rende il tipo task a tutti gli effetti un tipo privato e limitato.

Un oggetto di un tipo task può anche comparire nella lista dei parametri di un sottoprogramma: in questo caso il passaggio del relativo argomento sarà sempre ***per riferimento***.

Attivazione dei Tasks

Fin qui abbiamo assunto che un task si attivi quando l'unità che contiene la sua definizione raggiunge il **`begin`**.

Ci sono due fasi distinte che riguardano, invece, l'esecuzione di un oggetto task:
attivazione ? elaborazione delle dichiarazioni del corpo del task;
esecuzione ? esecuzione delle istruzioni.

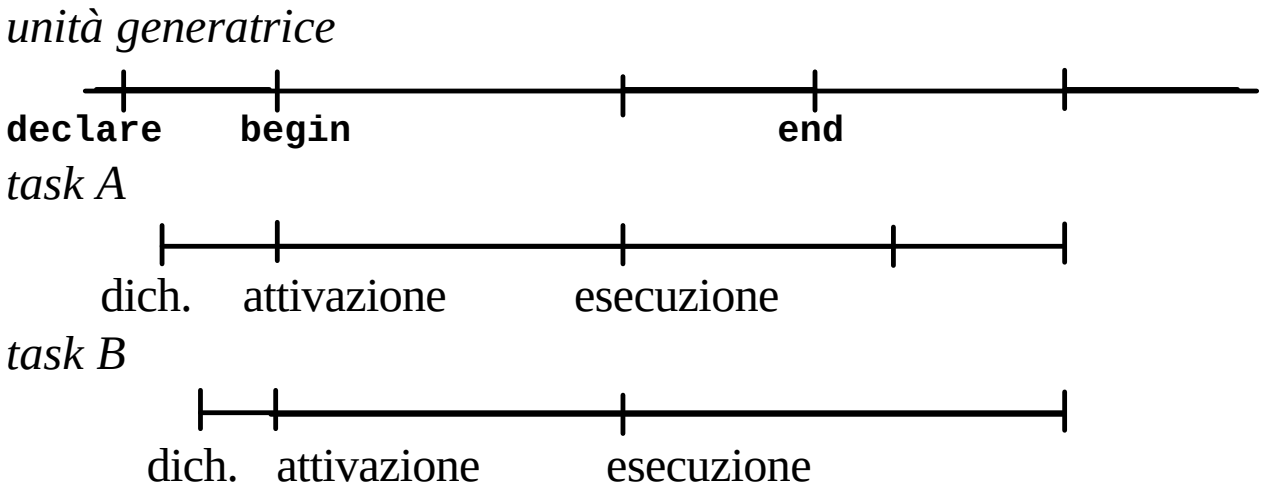
Durante la fase di attivazione l'unità che contiene la dichiarazione del task non può proseguire la sua esecuzione. Inoltre se sono presenti più task, la loro attivazione avviene in parallelo.

Quando l'unità che contiene la dichiarazione del task raggiunge la parola chiave **`begin`** essa inizierà la sua esecuzione in parallelo con i tasks in essa definiti.

Esempio

```
declare
     ....
     A : T;
     B : T;
     ....
begin
     ....
end;
```

Da un punto di vista grafico se si rappresenta con una linea continua l'intervallo di tempo in cui un task è attivo e con una linea tratteggiata l'intervallo in cui è inattivo, si ha:



Questo modo di gestire l'attivazione dei tasks è dovuto alla gestione delle eccezioni: si ricordi che le eccezioni generate durante la fase dichiarativa vengono riportate all'unità esterna e nel caso dei tasks vengono inviate all'unità generatrice sotto la forma di `TASKING_ERROR`. L'eccezione è inviata all'unità generatrice quando quest'ultima transita per il **`begin`**.

Se gli errori provengono da più tasks, una sola eccezione per tutti viene riportata all'unità originatrice; i tasks affetti da errore si completano senza arrecare disturbo agli altri.

Se un'eccezione è causata durante la fase dichiarativa dell'unità generatrice, i tasks dichiarati (non attivati) si concludono senza essere mai stati attivati.

Se un task è all'interno di un package, viene attivato al begin della sezione di inizializzazione del corpo del package. Se non c'è inizializzazione si considera una sezione con istruzione nulla; se invece non c'è proprio un corpo allora si considera un corpo composto di una sola sezione di inizializzazione con istruzione nulla.

Si osservi che i tasks creati con tipi accesso iniziano l'attivazione subito dopo la valutazione della funzione allocatore **new**, in qualsiasi punto essa si trovi.

Infine, le code delle entries di un task vengono create subito dopo la sua dichiarazione (prima dell'attivazione). Questo significa che già da questo momento gli altri tasks possono sospendersi su queste entries.

Conclusione dei Tasks

Il modo più semplice in cui un task può concludersi è quello di raggiungere l'**end**.
Se però un task contiene un ciclo infinito non raggiungerà mai l'**end**. Esiste allora una forma perticolare di **select**, che permette di inserire l'alternativa **terminate**.

Esempio

```
task body variabile_condivisa is
    var : elemento;
begin
    accept scrivi (x : in elemento) do
        v := x;
    end;
    loop
        select
            accept leggi (x : out elemento) do
                x := v;
            end;
        or
            accept scrivi (x : in elemento) do
                v := x;
            end;
        or
            terminate;
        end select;
    end loop;
end variabile_condivisa;
```

Il comportamento che seguirà il task `variabile_condivisa` è il seguente: se l'unità dalla quale dipende il task si è conclusa e se tutti i task collegati sono conclusi oppure sono in grado a loro volta di scegliere l'alternativa **`terminate`**, il task si conclude.

L'alternativa **`terminate`** può essere protetta da una guardia, ma non può essere abbinata a rami **`delay`** o **`else`**.

L'esecuzione di una **`terminate`** corrisponde ad una terminazione normale del task.

Accanto alla **`terminate`** esiste un'altra istruzione, la **`abort`**, che provoca la terminazione di un task. La sua sintassi è:

```
abort <lista_di_tasks>;
```

Esempio

```
abort T;
abort T1, T2, ... , Tn;
```

L'abort provoca un "effetto domino": quando viene forzata la terminazione di un task, vengono automaticamente costretti alla terminazione anche tutti i task che dipendono da questo, nonché gli eventuali sottoprogrammi o blocchi richiamati.

Se un task è sospeso viene fatto terminare immediatamente.
Qualche attenzione in più richiede invece la terminazione nel caso in cui il task non sia sospeso perché "*bisogna aspettare il momento più opportuno per far terminare il task*". Questa affermazione si riferisce ai casi in cui il task è partner di un rendez-vous:

- se il task è il richiamato, verrà fatto terminare ed al richiamante verrà segnalata l'eccezione di `TASKING_ERROR`;
- se il task è il richiamante, si attende la fine del rendez-vous ed a questo punto viene fatto terminare: nel frattempo è in uno stato ***anormale***.

La filosofia alla base è che il task richiamato solitamente è quello che offre un servizio ed una sua terminazione anormale condiziona pesantemente tutti i task che usufruiscono di detto servizio, i quali potrebbero essere costretti a terminare a loro volta proprio per il venir meno del servizio.

Il caso della terminazione del richiamante, cioè di un task che usufruisce di un servizio, non è importante perché il venir meno di un utente di un servizio non pregiudica in generale la bontà del servizio stesso; ma proprio per questo deve essere trattato con cura, in modo da evitare il crollo del richiamato e di conseguenza di tutti gli eventuali altri task che potrebbero richiamare quest'ultimo.

Una **`abort`** ha un effetto altamente distruttivo per cui si consiglia di usarla con molta attenzione e solo per casi eccezionali.

Lo stato di un task `T` può essere controllato attraverso gli attributi `TERMINATED` e `CALLABLE`:
`T'TERMINATED` vale `true` se il task `T` è terminato;
`T'CALLABLE` vale `true` se il task `T` non è completo, non è terminato e non si trova nello stato anormale di pre-terminazione.

Dall'analisi di questi attributi un task potrebbe decidere eventuali azioni da intraprendere. Per quanto riguarda l'attributo `CALLABLE`, però, c'è da dire che tra il momento in cui viene testato ed il momento in cui si intraprende un'eventuale azione il suo valore potrebbe cambiare: ad esempio il task testato potrebbe terminare e se l'azione conseguente fosse quella di stabilire un rendez-vous....