Departamento de Lenguajes y
Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

Apuntes para la asignatura

Informática

Facultad de Ciencias (Matemáticas)

http://www.lcc.uma.es/personal/pepeg/mates

Tema 7. Tipos de datos simples

Bibliografía

- *Programación 1*. José A. Cerrada y Manuel Collado. Universidad Nacional de Educación a Distancia.
- *Programming with TopSpeed Modula-2*. Barry Cornelius. Addison-Wesley.

Introducción

En este tema definimos el concepto de tipo de dato en un lenguaje de programación. También se clasifican los distintos tipos de datos que proporciona el lenguaje *Modula-2*. Además, se presentan las declaraciones de tipos que permiten al programador definir nuevos tipos de datos. En concreto, se estudian tres tipos de datos definibles por el programador: enumerados, subrangos y tipos subprogramas. Por último, se definen las reglas que sigue el compilador para detectar errores de tipo en un programa.

7.1 Concepto de tipo de datos. Clasificación.

Un *tipo de datos* es una descripción formal del conjunto de valores (o *dominio*) que una variable o expresión de dicho tipo puede tener, junto con el conjunto básico de operaciones que pueden ser aplicadas a estos valores.

En un lenguaje de alto nivel el concepto de tipo es de una gran importancia. En *Modula-2*, la declaración de una variable debe ir acompañada por la especificación de su tipo, ya que éste determina el espacio de memoria requerido para su almacenamiento y las combinaciones de operadores y operandos permitidos. El tipo de una constante puede ser deducido automáticamente por el compilador.

El aspecto práctico más importante de los datos es el modo en que pueden ser manipulados. Para ello, a cada tipo de datos se le asocia un conjunto de operadores básicos. La selección de estos operadores básicos es en cierta medida arbitraria, y podría haberse aumentado o disminuido. El criterio habitualmente seguido es seleccionar el conjunto mínimo de operadores que permita al programador construir cualquier operación de un modo razonablemente eficiente.

Los operadores más importantes definidos para cualquier tipo de dato son:

- La *asignación* (`:=`). Evalúa la expresión a su derecha y guarda el resultado en la variable a su izquierda.
- La *verificación de igualdad* (`=`). Comprueba si los valores a su izquierda y a su derecha son iguales.

Un mismo símbolo (por ejemplo los dos operadores anteriores) puede utilizarse como operador para distintos tipos de datos. Esto se denomina *sobrecarga*. EL operador + también está sobrecargado, ya que puede usarse para sumar valores de tipo INTEGER, CARDINAL y REAL (aunque los tipos de los dos argumentos han de ser compatibles).

7.1.1 Clasificación de tipos en Modula-2

Podemos clasificar los tipos que aparecen en el lenguaje *Modula-2* como:

- Tipos simples (escalares)
    - Ordinales
        - Predefinidos
            - `CARDINAL`, `LONGCARD`
            - `INTEGER`, `LONGINT`
            - `CHAR`
            - `BOOLEAN`
        - Definidos por el programador
            - Enumerados
            - Subrango
    - `REAL`, `LONGREAL`
- Tipos procedimiento y función
- Tipo estrucuturados
    - `ARRAY`
    - `RECORD`
    - `SET`
    - `FILE`
- `POINTER`

Todos los tipos simples son tipos *escalares*, ya que:

a) Están formados por elementos indivisibles

b) Están ordenados, es decir, se pueden comparar usando `=`, `<>`, `>`, ...

Todos los tipos simples, excepto los tipos reales, son *ordinales*:

c) Además de a) y b) cada valor (excepto el primero y el último) tiene un predecesor y un sucesor único.

Para todos los tipos ordinales podremos usar las siguientes operaciones:

- **`ORD`**`(X)` es una función que devuelve un CARDINAL (de cero en adelante), correspondiente al orden de `X`.
- **`VAL`**`(T,X)` es una función que dado un tipo `T` y un número de orden `X` devuelve el valor de tipo `T` con dicho orden. Nótese que **`ORD`**`(X)` = **`VAL`** `(`**`CARDINAL`**`, X)` `(X` de un tipo ordinal`)` y **`CHR`**`(X)` = **`VAL`**`(`**`CHAR`**`, X)` `(X` valor entero entre 0 y 255`)`.
- **`INC`**`(X)` es un procedimiento que asigna a la variable `X` el sucesor del valor que contenga. Es un error utilizar **`INC`**`(X)` si `X` contiene el último valor posible para el tipo.
- **`DEC`**`(X)` es un procedimiento que asigna a la variable `X` el antecesor del valor que contenga. Es un error utilizar **`DEC`**`(X)` si `X` contiene el primer valor posible para el tipo.
- **`INC`**`(X,N)` y **`DEC`**`(X,N)`. Aplican `N` veces **`INC`**`(X)` o **`DEC`**`(X)` respectivamente. N puede ser de tipo INTEGER o CARDINAL.
- **`MAX`**`(T)` es una función que devuelve el máximo valor posible para el tipo `T`.
- **`MIN`**`(T)` es una función que devuelve el mínimo valor posible para el tipo `T`.

Recordemos que la variable de control de un bucle FOR debe ser ordinal y que el tipo de la expresión en una sentencia CASE también.

Los operadores tienen que aplicarse a objetos de tipos de datos compatibles (iguales o no). La información del tipo de dato ayuda a los compiladores a detectar operaciones inapropiadas con tipos de datos no compatibles.

7.2 Tipos simples definidos por el programador

Aunque los tipos simples predefinidos vistos son útiles para cualquier propósito (que no requiera tipos estructurados), es conveniente que el programador pueda definir sus propios tipos simples por dos razones principalmente:

1) Interpretación de cada valor. Si una variable representa un número de naranjas y otra un número de personas, y representamos ambas variables con tipo CARDINAL podríamos sumar naranjas y personas, lo cual no tiene mucho sentido. Sería mejor poder definir un tipo NARANJAS y otro PERSONAS.
2) Dejar claro el rango de los posibles valores. Si sabemos a priori que para resolver cierto problema el valor de una variable nunca va a salirse de cierto rango, es conveniente declarar la variable con un nuevo tipo que se corresponda a dicho rango para que, en el caso de que por error tome un valor fuera de este rango, el compilador nos avise de ello.

La especificación de nuevos tipos se realiza usando una declaración de tipo. Por ejemplo:

```
TYPE COLOR = ...
VAR  Color1, Color2: COLOR;
```

Una declaración de tipo debe hacerse en la parte declarativa del programa antes de la declaración de variables de ese tipo y consiste en la palabra reservada TYPE seguida por el nombre del tipo, el signo igual y la especificación del tipo. En notación BNF:

Declaracion_de_Tipo	**::=**	TYPE Identificador_de_Tipo =	Tipo_Enumerado |
			Tipo__Subrango |
			Tipo_Función |
			Tipo_Procedimiento | ...
Identificador_de_Tipo	**::=**	Identificador	

7.2.1 Tipo enumerado

Consiste en una *enumeración* de todos los posibles valores que puede tomar una variable de dicho tipo encerrados entre paréntesis. Una variable de tipo enumerado sólo podrá tomar, en un determinado momento, uno de esos valores.

El uso de tipos enumerados ayuda a mejorar la legibilidad de los algoritmos cuando los valores de ciertas variables tienen una determinada interpretación. Si quisiéramos definir una variable para que represente una situación de un problema en el que hay cinco posibles colores, (Rojo, Amarillo, Verde, Azul y Naranja) podríamos hacer lo siguiente:

```
CONST
   Rojo     = 0 ;
   Verde    = 1 ;
   Azul     = 2 ;
   Amarillo = 3 ;
   Naranja  = 4 ;
VAR
   Color1, Color2 : CARDINAL ;
```

Sin embargo, en *Modula-2* podemos definir un nuevo tipo ordinal que conste de esos cinco valores exactamente:

```
TYPE COLOR = (Rojo, Verde, Azul, Amarillo, Naranja);
VAR  Color1, Color2 : COLOR;
```

La declaración de un tipo de esta índole consiste en asociar a un identificador una enumeración de los posibles valores que una variable de ese tipo puede tomar.

Otros ejemplos de tipos enumerados son:

```
TYPE
  PALO   = (Oros, Copas, Espadas, Bastos);
  DIA    = (Lunes, Martes, Miercoles, Jueves, Viernes,
            Sabado, Domingo);
  SEXO   = (Hombre, Mujer);
  FIGURA = (Circulo, Triangulo, Cuadrado, Rectangulo);
```

Un mismo valor NO PUEDE aparecer la declaración de DOS TIPOS ENUMERADOS distintos. Es decir, dada la declaración anterior la siguiente declaración NO es válida, ya que Rojo, Verde y Azul están usados en el tipo COLOR:

```
TYPE COLORBASICO = (Rojo, Verde, Azul);
```

El orden de los valores de estos nuevos tipos declarados por el programador será aquel en que aparecen dentro de la lista de enumeración de los elementos del tipo. El tipo enumerado es también escalar y ordinal.

La función **ORD** devuelve el número ordinal de un valor perteneciente a un tipo enumerado. La función **VAL** produce el efecto contrario:

Expresión	Valor
ORD(Rojo)	0
ORD(Azul)	2
VAL(COLOR,1)	Verde
VAL(COLOR,4)	Naranja

Al tratarse de tipos ordinales, los valores de tipo enumerado tienen su predecesor (excepto el primero del tipo) y sucesor (excepto el ultimo del tipo), por lo que es válido:

```
IF Color1 < Azul THEN ...
FOR Color1 := Rojo TO Azul DO...
CASE Color1 OF
  Rojo: ...
```

No existen operaciones aritméticas definidas para los tipos enumerados. Por lo tanto, es INVÁLIDO:

```
Color1 := Color1 + 1;
```

Sí se pueden utilizar, en cambio, los procedimientos generales **INC** y **DEC**.

Hay que tener en cuenta que se pueden producir errores al emplear los procedimientos **INC** y **DEC** si se excede el rango del tipo de la variable:

```
VAR  Color1  : COLOR;
     i       : INTEGER;
...
  Color1 := Rojo;
  i := 1000;
  INC (Color1);       ---> Color1 = Amarillo
  INC (i);            ---> i = i + 1 = 1001
  DEC (Color1, 4);    ---> Error
  INC (i,40000);      ---> Error
...
```

Los valores de tipo enumerado no pueden leerse ni escribirse directamente. Es tarea del programador el implementar los procedimientos y funciones adecuados. Por ejemplo el siguiente procedimiento puede ser usado para escribir un valor del tipo COLOR:

```
PROCEDURE WrColor(c : COLOR);
BEGIN
  CASE c OF
    Rojo     : WrStr (“Rojo”)   |
    Verde    : WrStr (“Verde”)  |
    Amarillo : WrStr (“Amarillo”)
    Azul     : WrStr (“Azul”)   |
    Naranja  : WrStr (“Naranja”)
  END
END WrColor;
```

La sintaxis BNF para la declaración de un tipo enumerado es:

Tipo_Enumerado	**::=**	(Lista_de_Identificadores) ;

Por último comentemos que el tipo BOOLEAN es realmente un tipo enumerado definido como:

```
TYPE BOOLEAN = (FALSE, TRUE);
```

para el cual han sido definidos otros operadores adicionales: **AND**, **OR** y **NOT**.

7.2.2 Tipo subrango

Siempre que podamos, debemos definir tipos cuyo rango de valores esté contenido en el rango de valores de otros tipos ordinales, ya que ayuda a clarificar el rango de valores que pueden tomar determinadas variables, evitando o descubriendo posibles errores. Los tipos con esta característica se llaman tipos *subrango*. Por ejemplo:

```
TYPE
  COLOR = (Rojo, Verde, Azul, Amarillo, Naranja);
  EDAD  = [0..130];     (* Subrango de CARDINAL *)
  NOTA  = [0..10];      (* Subrango de CARDINAL *)
  DIADELMES = [1..31]; (* Subrango de CARDINAL *)
  LETRAMINUSCULA = ['a'...'z'];  (* Subrango de CHAR *)
  ENTEROPEQUENYO = [-10..10];    (* Subrango de INTEGER *)
  COLORPRIMARIO  = [Rojo..Azul]; (* Subrango de COLOR *)

VAR  EdadAlumno: EDAD;
     NotaAlumno: NOTA;
```

Los valores de un tipo subrango deben ser un rango de valores CONSECUTIVOS pertenecientes a un tipo ya definido, al cual se le denomina *tipo base*. SÓLO tipos ORDINALES pueden ser tipos base.

En la definición del tipo se puede indicar el tipo base:

```
    Edad = [0..130];           (* Subrango de CARDINAL *)
    Edad = INTEGER [0..130];   (* Subrango de INTEGER  *)
```

pero se puede también omitir, pues el compilador puede calcularlo a partir del tipo de las expresiones en todos los casos, salvo si es incapaz de decidir si se trata de CARDINAL o INTEGER, en cuyo caso se toma por defecto el tipo CARDINAL.

Uno de los motivos para usar los tipos subrango o enumerado es detectar cierta clase de fallos en la que se dan valores fuera de rango. No obstante, por defecto el compilador no detecta este tipo de errores, por lo cual se producen resultados indefinidos en tiempo de ejecución de los programas. En *Top-Speed*, si queremos que lo haga hay que indicárselo expresamente con una

anotación en el programa. Es lo que se llama una *directiva para el compilador*, que se sitúa al principio del programa escribiendo (*#check (range=>on) *). El mismo efecto se consigue poniendo a **on** la opción correspondiente dentro del submenú **Runtime cheks** del menú **Project**. En caso de que en ejecución se presente un error de los que controla dicha directiva, aparecerá un mensaje de error indicándolo y preguntará si queremos ir al lugar del fichero fuente en el que se produjo. Hay que tener en cuenta que cuando está activa esta directiva la velocidad de ejecución decrece apreciablemente, por lo que normalmente sólo se usa en las etapas iniciales del desarrollo de un programa, de forma que cuando éste ha sido probado y suponemos que funciona se vuelve a compilar, pero sin indicar la directiva, con lo que el código objeto obtenido es más eficiente.

La sintaxis BNF para definir tipos subrango es:

Tipo_Subrango	**::=**	**[**Identificador_de_Tipo**]** [Expresión_Constante.. Expresión_Constante] ;

Las expresiones constantes deben tener tipo ordinal.

Como tipo ordinal que es, al tipo subrango también se le pueden aplicar las funciones que vimos en el primer apartado de este tema.

7.3 Compatibilidad de Tipos en Modula-2

Modula-2 es un lenguaje *fuertemente tipado*. Esto significa que el compilador chequea los tipos de los valores que aparecen en las sentencias y expresiones para comprobar el uso correcto de los mismos. Para ver las reglas que se siguen para ello definiremos previamente:

- *Tipos Equivalentes*: Se dice que dos tipos son equivalentes si tienen el mismo nombre, o están derivados del mismo nombre de tipo. Ejemplo:

```
TYPE
   T1 = REAL;
   T2 = T1;
   T3 = T1;
```

  Son todos tipos equivalentes.

  Se requiere que los tipos de los parámetros formales y reales sean equivalentes, si el paso de éstos es por referencia.

- *Tipos Compatibles*: Dos tipos T1 y T2 son compatibles, si y sólo si es cierta una de las siguientes condiciones:

  1) T1 y T2 son tipos equivalentes
  2) T2 es subrango de T1 o viceversa.
  3) T1 y T2 son subrangos del mismo tipo base.

  Modula-2 requiere compatibilidad de tipos para:

  1) Los operandos de una expresión aritmética.
  2) Los operandos de una expresión relacional.
  3) Las etiquetas y expresión de una sentencia CASE.
  4) El valor inicial, límite y variable de control de un bucle FOR.

- *Compatibilidad de asignación*: Dos tipos son compatibles en la asignación si son compatibles o si ambos son INTEGER o CARDINAL, o bien subrangos de los tipos INTEGER o CARDINAL. Ejemplo:

```
TYPE
  T1 = [0..10];
  T2 = [-3..3];
```

Modula-2 requiere este tipo de compatibilidad para:

1) La variable y la expresión de una sentencia de asignación.
2) El tipo del índice de un ARRAY y la expresión usada como índice (Tema 8).
3) Un parámetro por valor y su correspondiente formal.
4) EL tipo del resultado que devuelve una función y el valor que sigue al RETURN de dicha función.

7.4 Tipos procedimiento y función

En *Modula-2* es posible declarar variables de tipo subprograma (procedimientos o funciones) y asignar valores a éstas. Los valores asignables son nombres de subprogramas. Dichas variables pueden ser usadas como identificadores de subprogramas en una sentencia de llamada a procedimiento o función y el subprograma que se llame será el último que se haya asignado a la variable. Por ejemplo, el siguiente programa pide al usuario que elija una entre varias funciones, un punto x y evalúa la función en dicho punto:

```
MODULE Funciones
FROM IO IMPORT WrStr, WrLn, WrLngReal, RdLngReal, RdLn, RdChar;
FROM MATHLIB IMPORT Sin, Sqrt;
CONST
  PRECISION = 5;
TYPE
  FUNCION = PROCEDURE (LONGREAL) : LONGREAL;
VAR
  f      : FUNCION;
  x      : LONGREAL;
  Opcion : CHAR;

PROCEDURE Cuadrado ( X : LONGREAL) : LONGREAL;
BEGIN
  RETURN x*x
END Cuadrado;

PROCEDURE Cubo ( X : LONGREAL) : LONGREAL;
BEGIN
  RETURN x*x*x
END Cubo;

BEGIN
  WrStr (“Dame x:”);
  x := RdLngReal(); RdLn;

  WrStr (“1. Cuadrado”); WrLn;
  WrStr (“2. Cubo”); WrLn;
  WrStr (“3. Seno”); WrLn;
  WrStr (“4. Raiz cuadrada”); WrLn;

  WrStr (“Qué función: “);
  Opcion = RdChar(); RdLn;

  CASE Opcion OF
    ‘1’  : f := Cuadrado  |
```

```
      '2'  : f := Cubo      |
      '3'  : f := Sin       |
      '4'  : f := Sqrt
   END;

   WrStr ("El valor de la función en el punto es: ");
   WrLngReal (f(x), PRECISION, 0)
END Funciones.
```

La variable `f` es una variable de tipo función. En concreto se puede asignar a `f` el nombre de cualquier función que tome un valor `LONGREAL` y devuelva un valor `LONGREAL`. El tipo FUNCION es el tipo de todas las funciones con rango un valor `LONGREAL` y dominio un valor `LONGREAL`. No se puede asignar a `f` el nombre de otra función que no tenga este tipo. Por supuesto, podemos declarar distintos tipos funciones correspondientes a otras clases de funciones:

```
TYPE
   FUNCION2 = PROCEDURE (INTEGER, INTEGER) : INTEGER;
   FUNCION3 = PROCEDURE (CHAR) : INTEGER;
```

`FUNCION2` es el tipo de aquellas funciones que toman dos enteros y devuelven un entero. Similarmente `FUNCION3` es el tipo de las funciones que toman un carácter y devuelven un valor entero.

Podemos observar que se pueden asignar funciones importadas de otros módulos como son `Sin` y `Sqrt`, siempre y cuando sean del tipo adecuado. No podremos asignar nombres de subprogramas anidados. Sólo se pueden asignar nombres de subprogramas definidos en el nivel más externo o importados de otros módulos. Tampoco se pueden asignar nombres de operadores (+, - , ...).

Todo lo explicado para funciones vale para procedimientos. Tipos procedimientos válidos son:

```
TYPE
   PROCEDIMIENTO1 =PROCEDURE (INTEGER, INTEGER);
   PROCEDIMIENTO2 =PROCEDURE (CHAR);
```

La sintaxis BNF para los tipos función y procedimientos son:

Tipo_Procedimiento	**::=**	PROCEDURE **[**Lista_de_Tipos_Formales **];**
Tipo_Función	**::=**	PROCEDURE Lista_de_Tipos_Formales : Identificador_de_Tipo ;
Lista_de_Tipos_Formales	**::=**	(**[** Tipo_Formal **{** , Tipo_Formal **}** **]**)
Tipo_Formal	**::=**	**[** VAR **]** Identificador_de_Tipo

Relación de Problemas (Tema 7)

1. Escribe un programa que solucione el sistema de ecuaciones:

$$\begin{aligned} ax + by &= c \\ dx + ey &= f \end{aligned}$$

Para ello diseña un subprograma que tome como parámetros por valor `a`, `b`, `c`, `d`, `e` y `f` , como parámetros de salida por referencia `x`, `y` y `Estado`, donde `Estado` es del tipo enumerado:

```
TYPE Estado = (Determinado, Indeterminado, Incompatible);
```

Si el sistema está determinado `x` e `y` contendrán la solución al sistema y `Estado` valdrá `Determinado`. En otro caso `Estado` valdrá `Indeterminado` o `Incompatible` y se enviará un mensaje al usuario indicando el estado del sistema.

El programa principal deberá leer los coeficientes, llamar al subprograma y escribir el resultado por pantalla.

NOTA: Si $ae - bd = 0$ y $bf - ec = 0$ el sistema es indeterminado. Si $ae-bd = 0$ pero $bf-ec \neq 0$ el sistema es incompatible.

2. Escribir una función que tome como parámetros dos funciones `f` y `g` ambas de rango y dominio `LONGREAL`, un valor `x` de tipo `LONGREAL` y devuelva `f` compuesta de `g` evaluada en el punto `x`. Probar la función con casos concretos de `f` y `g`.

NOTA: $(f \circ g)(x) = f(g(x))$

3. Escribir un subprograma que tome como parámetro una función `f` de rango y dominio LONGREAL y un valor `x` de tipo LONGREAL, y devuelva un valor aproximado de la derivada de `f` en dicho punto utilizando para ello la siguiente fórmula con ε igual a una millonésima.

$$f'(x) = \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

Escribir un programa que utilice la función anterior para calcular el valor de la derivada de las funciones seno y coseno en los puntos π y π/2.

4. Escribir una función que encuentre un cero de una función utilizando el algoritmo de bipartición. Este algoritmo toma como parámetros, entre otros, una función `f` (de dominio y rango LONGREAL que suponemos continua), y tres valores `a`, `b` y `Epsilon` de tipo LONGREAL, y devuelve el cero calculado. El algoritmo debe comprobar que el valor de `b` sea mayor que `a` y que el signo de `f(a)` es distinto que el signo de `f(b)` (condiciones del Teorema de Bolzano). A partir de aquí se repite el siguiente proceso: calcular el punto medio del intervalo `[a,b]` y llamarlo `c`. Si `f(c)` = 0, entonces hemos encontrado un cero de `f` en `c` y acabamos. En otro caso determinamos en qué mitad del intervalo `[a,b]` se produce el cambio de signo (`[a,c]` o `[c,b]`) y repetimos el proceso con dicha mitad. Todo este proceso se repite hasta que se da la condición anterior (`f(c)` = 0) o hasta que el ancho del intervalo considerado sea menor que `Epsilon`. Utilizar la función anterior para encontrar un cero de la función coseno en [0, π].

5. Escribir una función `Pliega` que tiene cuatro parámetros: `f`, `a`, `b`, `Inicial`. El primer parámetro es una función que toma dos enteros y devuelve un entero. Los otros tres parámetros son valores enteros. El resultado de la función Pliega es de tipo entero y debe ser el resultado de calcular la expresión:

`f ( f ( ...f ( f ( f ( Inicial, a), a+1), a+2), ... b-1), b).`

Es decir , calculamos primero f con parámetros `Inicial` y `a` y obtenemos un resultado. Con este resultado y a+1 volvemos a calcular f. Seguimos así hasta que llegamos a que el segundo parámetro es b.

Escribir un programa que utilice la función `Pliega` para calcular el factorial de un número `N` leído de teclado y el sumatorio de los primeros `N` naturales, donde `N` es un número leído del teclado

6. Realizar un procedimiento `Filtra` que toma tres parámetros: `f`, `a` y `b`. `f` es una función que toma un entero y devuelve un booleano. `a` y `b` son dos valores enteros. Filtra deberá escribir por pantalla todos los valores enteros `x` en el intervalo [`a`..`b`] para los cuales `f(x)` sea cierto.

Utilizar el procedimiento `Filtra` en un programa que escriba los números perfectos y los números primos menores a un valor `Maximo` dado por teclado.

7. Escribir una función que calcule el valor de la integral definida

$$\int_a^b f(x)dx\,.$$

Esta función tomará como parámetros: una función `f` de dominio y rango LONGREAL, los valores `a` y `b` de tipo LONGREAL, y un parámetro `Epsilon` también LONGREAL. Para ello dividiremos el intervalo [`a`,`b`] en intervalos de amplitud menor o igual a `Epsilon` y aproximaremos la integral de cada subintervalo con la fórmula del trapecio:

$$\int_{x0}^{x1} f(x)dx \cong \frac{x1-x0}{2}\left[f(x0)+f(x1)\right]$$

El valor de $\int_a^b f(x)dx$ será el sumatorio de las aproximaciones anteriores. Utilizar la función anterior para calcular la integral definida de la función seno entre 0 y π/2.