uc3m

Basic Structs Language (BSL)

Curso 2019-2020

Especificaciones del lenguaje de programación BSL
Campus de Colmenarejo, curso 2019-2020

Contenido

Indice de tablas

1 Introducción

Este documento detalla la especificación y manual del lenguaje de programación BSL (Basic Structs Language). A lo largo de las siguientes páginas se introduce el lenguaje y con una descripción más formal de sus elementos, así como de las relaciones entre ellos. El manual está acompañado de ejemplos de código que permitirán aclarar lo explicado de forma teórica y pueden ser tomados de ejemplo para escribir programas nuevos.

El propósito es servir como herramienta de base para la enseñanza de asignaturas sobre procesadores del lenguaje, por lo que su tamaño y alcance ha sido intencionalmente restringido. Este manual abarca los datos necesarios para comprobar la legalidad de un programa, pero no incluye detalles sobre cómo generar código fuente una vez se ha identificado que la entrada es correcta.

No obstante, la especificación completa del lenguaje contiene información muy importante que no aparece en otras partes, por lo que es vital no perderse detalles que puedan llevar a una implementación incorrecta de las distintas comprobaciones léxicas, sintácticas y semánticas – y por tanto a dar por buena una entrada incorrecta, o a descartar como erróneo un programa bien formado.

2 ¿Qué es Basic Structs Language?

BSL (Basic Structs Language, variación del lenguaje VERYBasic) es un pequeño lenguaje de programación imperativo fuertemente tipado. El lenguaje trabaja con varios tipos de datos básicos y permite definir tipos compuestos de tipo estructura (o registros). Las estructuras de control básicas son el operador de salto condicional y el bucle condicional. Los tipos básicos disponibles son:

1. Números enteros
2. Números reales
3. Booleanos: verdadero y falso
4. Carácter: número en el rango 0-255. Puede interpretarse como un carácter de texto.

El tipo compuesto disponible es el registro de tipo struct

5. STRUCT nombre {tipo 1 campo1; tipo 2 campo2; .... tipo n campon;}

A continuación, mostramos un ejemplo de programa en lenguaje BSL. Nótese que hay dos tipos de comentarios:

Comentario de múltiples líneas, comenzando por "**<!--**" y finalizando en "**-->**".

Comentario hasta el fin de línea (o final de archivo si ocurre antes), comenzando por "**#**"

```
<!-----------------------------
---- Programa de ejemplo ------
-----------------------------
-- Declaracion de variables -->
REAL re1;
ENTERO en2 := 6; # con asignación de valor
BOOLEANO bo3 := 5<3; # se asigna el resultado de una expresión
CARACTER ca4 := 'h'; # variable de tipo carácter, y su literal
STRUCT BOLA {REAL cx; REAL cy; REAL radio}; #definición tipo compuesto
BOLA punto1; # declaración variable struct (no permite asignación al
#               mismo tiempo que la declaracion


### Expresiones
5 + 6; # operacion
re1 := 3.7; # asignación simple
bo3 := bo3 < 7 AND 5.46+7*en2 | 4; # asignación compleja
punto1.cx := 0.0; # campo de variable struct
punto1.cy := 0.0; # campo de variable struct
punto1.radio := 10.0; # campo de variable struct

### Sentencia condicional de control de flujo
SI en2==3 OR bo3 AND re1 # condición
ENTONCES # cuerpo cuando la condición es verdadera
    en2 := en2+3;
SINO # cuerpo cuando la condición es falsa
    re1 := 10-en2*en2;
FINSI # fin de la sentencia

### Sentencia de control de flujo bucle condicional
seguir := true;
MIENTRAS seguir AND punto1.radio>0 # bucle condicional
SI punto1.cx <0 AND punto1.cy <0 # cuerpo si condición verdadera
    seguir := false;
```

```
SINO # cuerpo cuando la condición es falsa
    seguir := true;
FINSI # fin de condicional
punto1.radio := punto1.radio/2.0;
punto1.cx := punto1.cx - 1.0;
punto1.cy := punto1.cy - 1.0;

FINMIENTRAS # fin de bucle MIENTRAS
```

Como hemos visto, será posible declarar y usar variables. El tipo de una variable debe ser especificado en su declaración, y no se permiten re-declaraciones (si la variable entera "en2" ha sido declarada una vez, no se podrá volver a declarar ninguna otra con el mismo nombre, tenga o no similar tipo).

El lenguaje proporciona operadores aritméticos, lógicos y de comparación. Los operandos pueden ser valores literales o variables indistintamente, aunque para poder combinarlos sus tipos de dato deben cumplir ciertas reglas (se mencionarán más adelante).

BSL incorpora un sistema de definición de funciones: se permite recursividad sin ningún tipo de restricción, y también pueden coexistir varias funciones con el mismo nombre siempre que su lista de parámetros sea única (lo que se conoce como "sobrecarga de parámetros"). A continuación, mostramos un ejemplo de programa incorporando funciones:

```
<!-----------------------------
---- Programa de ejemplo ------
-----------------------------
-- Declaracion de variables -->
### Definicion de Funcion
FUNCION cuadrado( REAL a ) RETURN REAL
{
    a * a;
}
### Declaracion de variables
REAL re1;
ENTERO en2 := 6; # con asignación de valor
BOOLEANO bo3 := 5<3; # se asigna el resultado de una expresión
CARACTER ca4 := 'h'; # variable de tipo carácter, y su literal
### Expresiones
5 + 6; # operacion
re1 := 3.7; # asignación simple
re1 := 3.7 + cuadrado(re1); # usando llamada a función
bo3 := bo3 < 7 AND 5.46+7*en2 | 4; # asignación compleja
te5 := te5 + " mundo"; # operadores adaptados a los tipos de dato
### Sentencia condicional de control de flujo
SI en2==3 OR bo3 AND re1 # condición
ENTONCES # cuerpo cuando la condición es verdadera
    en2 := en2+3;
SINO # cuerpo cuando la condición es falsa
    re1 := cuadrado( en2 );
FINSI # fin de la sentencia
### Sobrecarga de funciones
FUNCION cuadrado( entero e ) RETURN entero
{
    e * e;
}
```

En cuanto a la ejecución, el lenguaje no proporciona una estructura estándar para definir el punto de entrada al programa (lo que se conoce como "método main"), sino que todo el programa en sí mismo es un "main".

Las tres siguientes secciones proporcionan una descripción más formal del lenguaje, parte por parte. La primera sección describe los elementos desde el punto de vista morfológico, y se corresponde con la fase de análisis léxico.

La segunda parte es una descripción de cada elemento estructural del lenguaje y de las

directamente con el análisis sintáctico (y parte del semántico) se apoya en reglas expresadas usando notación BNF. Como nota aclaratoria, el uso de corchetes '`[]`' indica la opcionalidad del contenido.

La tercera y última parte describe las reglas semánticas a tener en cuenta para terminar de decidir si un programa es válido o no, y también para entender cuál es la interpretación correcta de un programa.

3 Elementos del lenguaje (análisis léxico)

A continuación, se describen los distintos elementos que pueden aparecer en un programa BSL. Esta descripción está realizada desde el punto de vista morfológico –es decir, lo que se correspondería con la fase de análisis léxico en un compilador.

En la introducción se mencionan los tipos de datos con los que trabaja BSL. Un determinado valor de cualquiera de estos tipos se puede definir explícitamente usando un "valor literal". Los literales de cada tipo tienen la forma que se resume a continuación:

Tabla 1 Valores literales del lenguaje BSL

	Morfología
ENTERO	Un literal del tipo numérico ENTERO se define como: • Un carácter numérico (0-9) • Una secuencia de más de un carácter numérico, cuyo primer elemento es distinto de cero ("015" no es un literal válido) **EJEMPLOS: 5, 26, 0, 4954862**
REAL	Número real en notación expandida o científica: • Notación expandida: número entero (ver definición anterior) seguido de un punto ".", y opcionalmente seguido de uno o más caracteres numéricos (0-9) **EJEMPLOS: 0.45, 15983.315, 54., 12.0** • Notación científica: número real con un único carácter distinto de cero a la izquierda del punto. Está seguido de un indicador de potencia de diez, construido como la letra "E" mayúscula, un signo opcional ("+" o "-") y un número entero indicando el exponente. Ejemplos: 5.4030E-10, 1.5E10, 8.E+10. **EJEMPLOS: 5.4030E-10, 1.5E10, 8.E+10**
BOOLEANO	Puede tomar dos valores (verdadero o falso), representados por sus equivalentes en inglés "true" y "false".
CARACTER	Cualquier carácter o símbolo encerrado entre comillas simples. **EJEMPLOS: 'a', '5', '.'**

Como se explica en la introducción, BSL permite definir y usar variables. Cualquiera de estos nombres se conoce como "identificador", y su forma legal es una cadena de caracteres compuesta por letras mayúsculas y minúsculas del alfabeto inglés, dígitos (0-9) y barras bajas ("_"). El primer carácter NO PUEDE ser un dígito.

Por lo tanto, tendremos que los siguientes identificadores serán válidos:

```
miCuadrado     cucu_tras   r3     __especial__      H54_B
```

Pero no así los siguientes:

```
Bla^5    54variable  ¿numero?    conTilde_á  Ñ_noValida
```

NOTA: *existen palabras reservadas que NO pueden usarse como identificador. Ver el final de esta sección.*

Aparte de estos elementos "variables", existen determinados símbolos con un significado especial. BSL define los siguientes símbolos:

Tabla 2 Símbolos del lenguaje BSL (no incluye operadores)

	Nombre	Descripción
;	Punto y coma	Indica el fin de determinadas sentencias
{ … }	Llaves (apertura, cierre)	Todo lo que se encuentra entre la llave de apertura y la llave de cierre se considera "agrupado" o perteneciente a un mismo bloque. Su aplicación se detalla en la sección de estructura del lenguaje, en concreto en 4.2 Sentencias
(…)	Paréntesis (apertura, cierre)	Parecido a las llaves, agrupa los elementos en su interior. Se usa para alterar el orden de evaluación en expresiones matemáticas.
:=	Asignación	Un símbolo "menor que" seguido de un guión. Su significado es asignar al elemento que hay a su izquierda el valor de lo que tiene en su parte derecha.
return	Salida	Su significado es declarar el tipo de salida de la función en su parte izquierda, que será lo que tiene en su parte derecha.

Existe otro tipo de símbolos, los operadores, que se usan para combinar o modificar valores y producir nuevos resultados. Los operadores de BSL se encuentran descritos en la siguiente tabla, aunque si significado concreto se especifica en la sección de reglas semánticas, *Combinación de valores*. Los operadores lógicos están duplicados y pueden representarse con símbolos o nombres (por ejemplo "And" y "&" devolverían el mismo token)

Tabla 3 Operadores del lenguaje BSL (binarios)

	Operador	Descripción
Aritmético	**+**	Símbolo "más". Suma valores numéricos
	-	Guión. Resta valores
	*****	Asterisco. Multiplica valores
	/	Barra de división. Divide valores numéricos
Comparación	**<**	"Menor que"
	<=	"Menor o igual que"
	>	"Mayor que"
	>=	"Mayor o igual que"
	==	Comprueba que dos elementos son iguales
Lógico	**And** **&**	Evalúa a verdadero si sus dos operandos son verdaderos.
	Or **|**	Evalúa a verdadero si al menos un operando es verdadero
	Not **!**	Invierte el valor del operando a la derecha

Es importante ver que los operadores lógicos no son símbolos en el sentido estricto de la palabra, sino más bien palabras. Estas dos palabras junto con otras cuantas que tienen un significado particular son lo que se conoce como "palabras reservadas".

Por su condición especial las palabras reservadas no pueden usarse como nombres de variables. Las palabras reservadas del lenguaje BSL son:

Tabla 4 Palabras reservadas del lenguaje BSL

True	Booleano	Entonces
False	Struct	Sino
Entero	Caracter	Finsi
Real	Mientras	And
Finmientras	Si	Or
Not	Return	Funcion

IMPORTANTE: *las palabras reservadas pueden escribirse en mayúscula, minúscula o cualquier combinación de ellas. Esto quiere decir que para declarar una estructura podrán usarse indistintamente "STRUCT", "struct", "Struct", "sTrUct", etc.*

4 Estructura del lenguaje (análisis sintáctico)

Esta sección contiene una descripción detallada de cada uno de los elementos estructurales del lenguaje, así como de la relación entre ellos. Al final de la sección se resume la sintaxis del lenguaje mediante una gramática en notación BNF.

Los elementos serán detallados comenzando por el Programa (que es el más amplio de todos), y a partir de ahí se irán analizando sus componentes.

4.1 Programa y bloques de sentencias

Como se ha mencionado anteriormente, el Programa es el elemento más general y amplio de nuestro lenguaje: cualquier fragmento de código bien formado es un programa. Si se juntan dos programas, el resultado es UN programa más grande.

Un programa, no obstante, es un bloque de sentencias. En la gramática que especifica el lenguaje se definen estos elementos como:

```
programa          ::= blq_sentencias
                  |   /*lambda*/
blq_sentencias    ::= blq_sentencias sentencia
                  |   sentencia
```

Podemos ver que un bloque de sentencias no es más que una sucesión simple de AL MENOS UNA sentencia. La entidad bloque de sentencias es importante, puesto que aparecerá en otros puntos de la gramática que define el lenguaje.

4.2 Sentencias

Cada una de las sentencias que componen un programa puede ser de varios tipos. En nuestro caso vamos a diferenciar entre:

1. Sentencias de declaración: definen nuevos elementos (variables, tipos).
2. Sentencias de uso: usan elementos anteriormente definidos (asignación, expresión).
3. Sentencia de control de flujo: BSL define el salto condicional como la única herramienta para controlar el flujo del programa

O, en notación BNF:

```
sentencia   ::= sent_decl
            |   sent_uso
            |   sent_flujo
```

Donde:

```
sent_decl   ::= decl_variable PCOMA
sent_uso    ::= asignacion PCOMA
            |   expresion PCOMA
flujo       ::= condicional
            |   bucle
```

Esta clasificación es sólo una de las múltiples posibilidades (ni siquiera es la más "formal"), pero tiene la ventaja de estar claramente estructurada y además es válida para el propósito de BSL.

Adicionalmente, y como sucede en otros lenguajes como Java o C, es posible agrupar varias sentencias en una sola encerrándolas entre llaves:

```
sentencia   ::= LLAVE_IZQ blq_sentencias LLAVE_DER
```

Esto sirve para crear bloques de código cuya utilidad se verá en la definición de la sentencia de control de flujo.

4.3 Expresiones

Comencemos las sentencias de uso por las expresiones. En nuestro lenguaje, una expresión es o bien una expresión binaria (toma dos valores como entrada y devuelve uno a su salida), o bien un término –operando– con valor final. Esto último quiere decir que cualquier literal o nombre de variable se interpreta como un valor (esto se ve en la sección *4.4 Operandos*, página 12).

Las expresiones serán de tipo aritmético (suma, resta, multiplicación, división), lógico (and, or) o de comparación (menor que, mayor que, o igual). Las expresiones deben terminar en punto y coma.

4.3.1 Orden de operaciones

En las expresiones, las operaciones siguen un orden establecido por la prioridad de los operadores. A igualdad de prioridad (esto es, concatenación de operaciones cuyos operadores tienen la misma prioridad), las sentencias se evalúan de izquierda a derecha.

Es importante notar que los operadores de comparación MAYOR, MENOR e IGUAL deberían ser NO asociativos y por tanto no pueden aparecer concatenados. Esto se traduce en que no serán legales sentencias del tipo:

```
5 < 4+3 > 8
```

Aunque mediante agrupación con paréntesis sería sintácticamente válido:

```
(5 < 4+3) > 8 # o su variante 5 < (4+3 > 8)
```

Sin embargo, y como veremos más adelante, este tipo de sentencias serán semánticamente incorrectas debido a conflictos con los tipos.

4.4 Operandos

Los operandos son aquellos términos que representan en sí mismos un valor. A saber, podemos encontrarnos con valores literales y variables.

Los literales son fragmentos de la entrada que representan explícitamente un valor un tipo básico:

```
operando    ::= ENTERO
            |   REAL
            |   BOOLEANO
            |   CARACTER
```

Las variables se usan escribiendo su nombre:

```
operando    ::= ID
```

y por ultimo, para variables de tipo STRUCT, podemos acceder a los campos existentes con el operador ".":

```
operando    ::= ID.ID
```

donde el segundo identificador debe haberse especificado en la declaración del tipo STRUCT correspondiente a la variable

4.5 Control de flujo

La primera sentencia de control de flujo del lenguaje es el salto condicional (equivalente al 'if' de otros lenguajes). Se describe a continuación

4.5.1 Salto condicional

El salto condicional es una sentencia que evalúa una expresión booleana –llamada condición–, y dependiendo de si su resultado es verdadero o falso, salta a un punto más avanzado del programa o sigue ejecutando las sentencias adyacentes.

Las sentencias BNF que describen la construcción de un salto condicional son las dos siguientes:

```
condicional ::= SI expresion ENTONCES blq_sentencias FINSI
            |   SI expresion ENTONCES
                    blq_sentencias
                SINO
                    blq_sentencias
                FINSI
```

Donde SI, ENTONCES, SINO y FINSI son palabras reservadas del lenguaje.

La primera variante descrita es un salto simple. Si la expresión de la condición evalúa a verdadero, se ejecutará el bloque de sentencias encerrado entre las palabras ENTONCES y FINSI y después se continuará con normalidad. En caso contrario, se saltará directamente a la primera instrucción a partir del fin del salto condicional.

La segunda opción es algo distinta. Si la condición evalúa a verdadero, se ejecutará el bloque de sentencias entre ENTONCES y SINO, y después se saltará a la sentencia siguiente al salto condicional. Si la condición evalúa a falso, se ejecutará el bloque entre SINO y FINSI, y luego se continuará con normalidad.

Nótese que las sentencias condicionales NO terminan en punto y coma

4.5.2 Bucle condicional

El bucle condicional es una sentencia que evalúa una expresión booleana –llamada condición–, y dependiendo de si su resultado es verdadero o falso, ejecuta el cuerpo del bucle o bien salta al final de éste. Las sentencias BNF correspondientes al bucle condicional son:

```
bucle ::= MIENTRAS expresion blq_sentencias
          FINMIENTRAS
```

Siendo MIENTRAS, FINMIENTRAS palabras reservadas del lenguaje.

De nuevo, la expresión condicional NO terminan en punto y coma.

4.6 Variables – declaración y uso

La declaración de una variable es su primera aparición en el programa. Para declarar una variable es necesario indicar su tipo, el nombre que se desee darle, y opcionalmente un valor inicial:

```
decl_variable ::= KEYTIPO ID ':=' expresion
              |   KEYTIPO ID
```

KEYTIPO debe ser un tipo de dato válido de BSL, que se enumeran en la introducción de este documento de especificación.

Aunque estas reglas de la gramática no lo expresen, las declaraciones de variable deben terminar en punto y coma, tanto si incluyen asignación de valor como si no, tal y como se indica en *4.2 Sentencias*.

Una variable que ya ha sido declarada puede ser usada en cualquier expresión. El significado es que se sustituye su aparición por el valor que tenga en ese punto del programa.

4.7 Tipos STRUCT

Pueden declararse variables de tipo "registro" o "estructura", que en BSL se denominan STRUCT. Como se ha indicado, es condición necesaria que previamente se haya declarado este tipo, para poder después declarar la variable:

```
decl_struct ::= STRUCT ID {lista_decl}
lista_decl  ::= decl_variable ; lista_decl
             | decl_variable ;
```

De nuevo, las declaraciones de variables dentro de la estructura deben terminar en punto y coma.

Ahora, la variable puede declararse como una estructura, no estando permitida la asignación en la misma declaración:

```
decl_variable ::= ID ID
```

donde el primer ID obligatoriamente debe corresponderse con el identificador de un tipo válido STRUCT previamente declarado con la sentencia anterior.

IMPORTANTE: *Obsérvese que con esta definición puede darse la "recursividad" en la construcción de tipos STRUCT*

4.8 Asignación

Una asignación es la operación por la cual se ajusta el valor de una variable al resultado de una expresión especificada.

La sentencia para describir asignaciones es:

```
asignacion ::= ID ':=' expresion
```

y en el caso de estructrucas, se utiliza el operador "." Para acceder al campo:

```
asignacion ::= ID.ID ':=' expresion
```

Donde ID representa la variable que recibe el valor de la expresión a la derecha.

Aunque esta regla de la gramática no lo exprese, las asignaciones deben terminar en punto y coma como se indica en *4.2 Sentencias* (página 10).

IMPORTANTE: *hay que fijarse en que por cómo esta gramática está construida, BSL no permite asignaciones encadenadas de tipo* `var1 := var2 := … := expresión;`

4.9 Declaración de funciones

Una función es un bloque de sentencias que, dados unos parámetros de entrada, realizan una serie de cálculos para obtener un único valor al que se denomina ***de retorno***.

Las funciones tienen un nombre que permite que sean referenciadas desde cualquier punto de un programa (de igual forma que se referencian las variables). Por tanto, para declarar una función es necesario especificar su nombre, el tipo y nombre de sus valores de entrada –argumentos–, el tipo de dato que tendrá su valor de retorno, y las sentencias que realizan los cálculos necesarios –cuerpo de la función–.

En nuestro lenguaje de programación, los argumentos no son obligatorios: una función puede recibir cero valores de entrada. Sin embargo, el cuerpo de una función debe contener, por lo menos, una sentencia.

Por ello, las sentencias que definen la declaración de una función son:

```
def_funcion ::= FUNCION ID '(' lista_args ')' 'RETURN' KEYTIPO
                '{' blq_sentencias '}'
```

Donde FUNCION y RETURN son palabras reservadas, ID es el nombre que recibirá la función, KEYTIPO es el nombre del tipo de dato que devuelve la función, y el cuerpo de la función es un bloque de sentencias encerrado entre llaves.

Sobre la lista de argumentos, se define como una secuencia de argumentos separados por coma, pudiendo ser válida la lista vacía. Un argumento es una pareja compuesta por un tipo de dato y un nombre, similar a una declaración de variable sin asignación inicial de valor:

4.10 Uso de funciones

Una función es invocada cuando se escribe su nombre, seguida de una lista de expresiones entre paréntesis que coincide en número y tipo con sus argumentos:

```
uso_funcion       ::= ID '(' lista_expresiones ')'
lista_expresiones ::= lista_expresiones ',' expresion
                  |   expresion
```

El resultado será un valor con el tipo de dato de retorno de la función. Es importante notar que cada uno de los argumentos de la función puede ser una expresión completa, siendo legales sentencias como:

```
bla( 5.0, (8.0+b)*cubo(b)<7 );
```

En este caso se invocaría a la función `bla( real, booleano )`, o a la que coincida en nombre y número de argumentos, y tenga tipos de argumentos compatibles más cercanos a `(real, booleano)`.

5 Reglas semánticas

Esta sección describe reglas adicionales sobre el lenguaje que no aparecen descritas en las secciones anteriores. Estas reglas aparecen organizadas por categoría conceptual (control de tipos, declaración y vida de símbolos…) en lugar de estar agrupadas según elemento del lenguaje al que se refieran –como sucedía en la sección anterior.

5.1 Reglas de tipos

Podemos agrupar las reglas de tipado en dos categorías:

- Movimiento de datos de un punto (entrada) a otro (salida) donde se usa:

```
real a := 5.3; # asignación de un valor a la variable "a"
SI true ENTONCES … # evaluación de una condición de salto
```

- Varios datos que se combinan para producir un resultado (otro dato):

```
5 * 8.2; # multiplicación: hay que saber cómo se combinan
         # estos valores y qué resultado producen
```

Ambos niveles están íntimamente relacionados, pues a veces es necesario resolver una operación que combina dos datos, para saber el tipo de dato que se va a producir es válido para una aplicación posterior:

```
entero b := 5 + 8.2;  # es necesario conocer el tipo de dato que genera
                      # la suma para saber si la asignación es legal
```

Detallamos a continuación cada uno por separado:

5.1.1 Movimiento, evaluación y conversión de datos

En todas las sentencias que mueven valores de un lugar a otro (p.ej. asignaciones), o que evalúan una expresión para usarla en un punto concreto (p.ej condiciones de salto), tanto el origen como el destino deben tener el mismo tipo de dato, o bien debe poderse realizar una conversión del tipo origen al tipo destino sin pérdida de datos.

En todas las sentencias que mueven valores de un lugar a otro (p.ej. asignaciones), o que evalúan una expresión para usarla en un punto concreto (p.ej argumentos de funciones o condiciones de salto), tanto el origen como el destino deben tener el mismo tipo de dato, o bien debe poderse realizar una conversión del tipo origen al tipo destino sin pérdida de datos.

Algunos tipos pueden transformarse automáticamente a otros, según la siguiente tabla de conversión:

Tabla 5 Conversión legal automática de tipos (sólo se incluyen conversiones directas)

Carácter	Entero	Se toma el valor número del carácter (0-255)
Entero	Real	Pasar de 32 bits que representan un entero en Ca2 a 32 bits que representan un real en IEEE754

De esta forma, las siguientes sentencias serán válidas:

```
real      r1 := 7.5;        # OK: real -> real
entero    e1 := 'a';        # OK: caracter -> entero
real      r2 := e1;         # OK: entero -> real
```

Pero no así los siguientes ejemplos:

```
entero     e1 := 7.5;          # ERROR: real->entero
booleano   b1 := 7;            # ERROR: entero -> booleano
booleano   b2 := r1;           # ERROR: real->bool (r1 es de tipo real)
```

5.1.2 Combinación de valores

Todas las operaciones que combinan/manejan dos o más valores deben asegurar que sus tipos son similares, o al menos que se puede realizar la conversión de uno de ellos al tipo del otro sin pérdida de datos.

Respecto a las operaciones y expresiones, cada operador requiere uno varios tipos de dato concretos a la entrada, y devuelve así mismo un tipo específico de dato a la salida. En la siguiente tabla se especifican los tipos de datos compatibles con cada operador (sin tener en cuenta conversiones), y la salida correspondiente para cada tipo de entrada.

Tabla 6 Operadores y tipos de datos

	Tipo Origen	Tipo Resultado
MAS	Entero	Entero
	Real	Real
	Carácter	Carácter
MENOS	Entero	Entero
	Real	Real
	Carácter	Carácter
POR, ENTRE	Entero	Entero
	Real	Real
MAYOR, MENOR	Entero	Booleano
	Real	Booleano
	Carácter	Booleano
IGUAL	Entero	Booleano
	Real	Booleano
	Carácter	Booleano
	Booleano	Booleano
AND, OR	Booleano	Booleano

No obstante, cuando una de las entradas de un operador binario tiene un tipo A, y la otra entrada un tipo B, será necesario convertir una de ellas a la otra: el dato con tipo más restrictivo se transforma al tipo más general. Así, **Carácter** puede convertirse a **Entero** o **Real**, y **Entero** puede convertirse a **Real**. No se permite la conversión de **Booleano** a ningún otro tipo.

A continuación, se muestran varios ejemplos donde se combinan todas las reglas referentes al tipado. Las siguientes sentencias son válidas en BSL:

```
5.0*4.0;                 # OK: real * real -> real
real r1 := 7.5+'c'*8;    # OK: por el orden de operación, primero se
                         # ejecuta 'c'*8 [carácter*entero->entero] y
                         # después la suma 7.5+<<resultado anterior>>
                         # [real+entero->real]
```

Pero no así estas otras líneas:

```
a := 5.0 * ('d'<8);      # ERROR: aunque la primera operación es válida
                         # 'd'<8 [carácter<entero, carácter se convierte
                         # a entero y se comparan], su resultado es un
```

```
                        # booleano, y 5.0*<<resultado>> [real*booleano]
                        # es ilegal porque real no puede convertirse
                        # directamente a booleano, ni al revés
entero e1 := 'c'+'c';   # OK: carácter -> entero
SI 5/7 ENTONCES;        # ERROR: real -> booleano
```

5.2 Símbolos – variables

Las variables y tipos son entidades definidas por el usuario. Esto quiere decir que para poder usar una determinada variable, o un tipo de estructura, primero deben haber sido "declaradas" en el programa.

No obstante, las reglas específicas que regulan la creación y vida de unas y otras difieren entre sí. Esta sección cubre todo lo necesario para determinar la legalidad de expresiones que involucren alguno de estos símbolos.

5.2.1 Variables – declaración y uso

Las variables son entidades locales sujetas a una línea temporal:

- Locales: una variable declarada en un determinado bloque de código sólo existe dentro de dicho bloque.
- Sujetas a una línea temporal: la vida de una variable comienza en el preciso instante en que es declarada, y su valor instantáneo depende de las sentencias anteriores en las que ha estado involucrada. Esto implica, entre otras cosas, que una variable sólo puede usarse DESPUÉS de haber sido declarada.

Como se ha visto anteriormente, una declaración requiere al menos especificar su tipo y el nombre que se le otorga:

```
real miVariable;
```

No se permiten identificadores repetidos. Es decir, después de la sentencia anterior, no sería legal la aparición de ninguna de las dos siguientes :

```
real    miVariable; # repetida
entero  miVariable; # mismo identificador, aunque tenga otro tipo
```

IMPORTANTE: *los nombres de las variables SON SENSIBLES A MAYÚSCULAS. Esto quiere decir que será legal declarar las siguientes cuatro variables en un mismo bloque, puesto que sus nombres NO SON EQUIVALENTES:*

```
real        miVariable;
entero      MIVARIABLE;
entero      MiVaRiAbLe;
```

Cuando una variable es declarada sin asignársele un valor inicial, se le asignará el valor por defecto para su tipo de dato, de acuerdo a la siguiente tabla. Como regla general, los valores numéricos reciben un valor equivalente a cero (incluyendo el carácter):

Tabla 7 Valores por defecto para variables sin inicializar

	Valor
Entero	0
Real	0.0
Booleano	False
Carácter	0 (no imprimible)

IMPORTANTE: *cuando se ejecuta una declaración de variable con inicialización:*

```
real miVariable := 5.0 + otraVarReal;
```

se asume que en primer lugar se resuelve la expresión de la inicialización, y a continuación se declara la variable. Esto quiere decir que expresiones como la siguiente no serían válidas:

```
real X := 5.0 + X;
```

5.2.2 Funciones – declaración y uso

Las funciones en BSL son símbolos globales y persistentes:

- Globales: una función declarada legalmente puede ser usada en CUALQUIER PARTE de un programa, independientemente del lugar de declaración (esto incluye el propio cuerpo de la función, algo necesario para habilitar la posibilidad de recursión directa).
- Persistentes: una vez una función es declarada, permanece disponible durante la vida completa del programa. No existen situaciones en las que la función deje de existir o no esté disponible.

Esto quiere decir, entre otras cosas, que las funciones pueden ser usadas en el programa ANTES de su declaración. Por ejemplo, el siguiente código es legal:

```
entero a := miFuncion( 5 );
Funcion miFuncion( entero x ) return entero { ... }
```

Sobre sus reglas concretas de declaración, es importante notar que BSL admite sobrecarga de funciones por cambio de parámetros. Es legal que existan dos funciones con el mismo nombre, siempre que su lista de parámetros varíe en número y/o tipo:

```
Funcion bla( entero a, real b ) return real { ... }
<!--Las dos siguientes definiciones son legales, porque su lista
 de parámetros es distinta de la 1a en número o en tipo -->
Funcion bla( entero a ) := real { ... }
Funcion bla( entero a, entero b ) return real { ... }
<!—Pero la siguiente definición es ilegal porque coincide con la 1a
 definición en nombre y parámetros. Da igual que el tipo de retorno
 sea distinto -->
Funcion bla( entero a, real b ) return booleano { ... }
```

Como se puede ver en la última línea del ejemplo, el tipo de retorno no influye en determinar si la función ya ha sido declarada o no.

IMPORTANTE: *los nombres de las funciones SON SENSIBLES A MAYÚSCULAS. Esto quiere decir que será legal declarar las siguientes funciones, puesto que sus nombres NO SON EQUIVALENTES:*

```
Funcion nombredefuncion() return real {…}
Funcion NombreDeFuncion() return real {…}
Funcion NOMBREDEFUNCION() return real {…}
Funcion nOMbrEDefuNCion() return real {…}
```

La sobrecarga de funciones hace que a veces pueda ser complicada saber qué función de las declaradas se va a corresponder con una determinada sentencia de uso. Supongamos que tenemos un fichero con las siguientes funciones declaradas:

```
# Varias declaraciones de funciones sobrecargadas
Funcion bla( entero a, real b ) return real { ... }        # función 1
Funcion bla( entero a ) return real { ... }                # función 2
Funcion bla( entero a, entero b ) return real { ... }      # función 3
Funcion bla( entero a, booleano b ) return real { ... }    # función 4
Funcion bla( real a ) return real { ... }                  # función 5
```

Y donde tenemos las siguientes funciones de uso:

```
# Uso de las funciones anteriores
bla( 5, 4.3 );    # compatible con 1 (sin conflicto)
bla( 3.0 );       # compatible con 5 (sin conflicto)
bla( 'd' );       # compatible con 2 (sin conflicto; carácter->entero)
bla( 4 );         # compatible con 2 y 5 (entero->entero o real)
bla( 5, 4|3 );    # compatible con 1 (entero, real -> real o booleano)
```

En los tres primeros casos, vemos que no hay conflicto, puesto que sólo existe una función con ese nombre y una lista de parámetros compatible (donde el tipo de cada parámetro cuadra con el de la declaración, o donde el tipo de dato en la sentencia de uso puede convertirse directamente al de la sentencia de declaración). Tendrá prioridad la llamada que no requiera ninguna conversión automática de tipos, de manera que solamente cuando la invocación no sea compatible con ninguna función con los mismos tipos de los argumentos, se intentará la conversión para otras funciones con tipos resultado de la conversión.