

## Using SDL Data Types



In this section, an overview is given of the data types that are available in SDL. SDL contains a number of predefined data types. Based on these predefined types it is possible to define user-specific data types. Types, or according to SDL terminology, "sorts", are defined using the keywords `newtype` and `endnewtype`.

### Example 1 : Newtype definition

```
newtype example1 struct
a integer;
b character;
endnewtype;
```

A newtype definition introduces a new distinct type, which is not compatible with any other type. So if we would have another newtype `otherexample` with exactly the same definition as `example1` above, it would not be possible to assign a value of `example1` to a variable of `otherexample`.

It is also possible to introduce types, *syntypes*, that are compatible with their base type, but contain restrictions on the allowed value set for the type. Syntypes are defined using the keywords `syntype` and `endsyntype`.

### Example 2 : Syntype definition

```
syntype example2 = integer
constants 0:10
endsyntype;
```

The syntype `example2` is an integer type, but a variable of this type is only allowed to contain values in the specified range 0 to 10. Such a constant clause is called a *range condition*. The range check is performed when the SDL system is interpreted. Without a range condition a syntype definition just introduces a new name for the same sort.

For every sort or syntype defined in SDL, the following operators are always defined:

- ≠ := (assignment)
- ≠ = (test for equality)
- ≠ /= (test for non-equality)

These operators are not mentioned among the available operators in the rest of this section. Operators are defined in SDL by a type of algebra according to the following example:

```
"+" : Integer, Integer -> Integer;
num : Character -> Integer;
```

The double quotes around the + indicate that this is an infix operator. The above + takes two integer parameters and returns an integer value. The second operator, `num`, is a prefix operator taking one Character and returning an Integer value. The operators above can be called within expressions in, for example, task statements:

```
task i := i+1;
task n := num('X');
```

where it is assumed that `i` and `n` are integer variables. It is also allowed to call an infix operator as a prefix operator:

```
task i := "+"(i, 1);
```

This means the same as `i := i+1`.

## Predefined Sorts

The predefined sorts in SDL are defined in an appendix to the SDL Recommendation Z100. Some more predefined sorts are introduced in the Recommendation Z105, where it is specified how ASN.1 is to be used in SDL. These types should not be used if the SDL system must conform to Z.100. The SDL suite also offers Telelogic-specific operators for some types. These operators should not either be used if your SDL system must be Z.100 compliant. The rest of this chapter describes all predefined sorts. Unless stated otherwise, the sort is part of recommendation Z.100.

### Bit

The predefined `Bit` can only take two values, 0 and 1. `Bit` is defined in Z.105 for the definition of bit strings, and is not part of Z.100. The operators that are available for `Bit` values are:

```
"not" : Bit -> Bit
"and" : Bit, Bit -> Bit
"or"  : Bit, Bit -> Bit
"xor" : Bit, Bit -> Bit
"=>" : Bit, Bit -> Bit
```

These operators are defined according to the following:

- ⌘ `not` :  
inverts the bit; 0 becomes 1 and 1 becomes 0,  
`not 0 gives 1, not 1 gives 0`
- ⌘ `and` :  
if both parameters are 1, the result is 1, else it is 0,  
`0 and 0 gives 0, 0 and 1 gives 0, 1 and 1 gives 1`
- ⌘ `or` :  
if both parameters are 0, the result is 0, else it is 1,  
`0 or 0 gives 0, 0 or 1 gives 1, 1 or 1 gives 1`
- ⌘ `xor` :  
if parameters are different, the result is 1, else it is 0,  
`0 xor 0 gives 0, 0 xor 1 gives 1, 1 xor 1 gives 0`
- ⌘ `=>` (implication) :  
if first parameter is 1 and second is 0, the result is 0, else it is 1,  
`0 => 0 gives 1, 1 => 0 gives 0, 0 => 1 gives 1, 1 => 1 gives 1`

The `Bit` type has most of its properties in common with the `Boolean` type, which is discussed below. By replacing 0 with `false` and 1 with `true` the sorts are identical.

Bit and Boolean should be used to represent properties in a system that can only take two values, like on - off. In the choice between Bit and Boolean, Boolean is recommended except if the property to be represented is about bits and the literals 0 and 1 are more adequate than `false` and `true`.

## Bit\_string

The predefined sort `Bit_string` is used to represent a string or sequence of Bits. `Bit_string` is defined in Z.105 to support the ASN.1 `BIT STRING` type, and is not part of Z.100. There is no limit on the number of elements in the `Bit_string`.

The following operators are defined in `Bit_string`:

```

mkstring : Bit -> Bit_string
length : Bit_string -> Integer
first : Bit_string -> Bit
last : Bit_string -> Bit
"/" : Bit_string, Bit_string -> Bit_string
substring : Bit_string, Integer, Integer
-> Bit_string
bitstr : Charstring -> Bit_string
hexstr : Charstring -> Bit_string
"not" : Bit_string -> Bit_string
"and" : Bit_string, Bit_string -> Bit_string
"or" : Bit_string, Bit_string -> Bit_string
"xor" : Bit_string, Bit_string -> Bit_string
"=>" : Bit_string, Bit_string -> Bit_string

```

These operators are defined as follows:

- ✎ `mkstring` :

This operator takes a Bit value and converts it to a `Bit_string` of length 1.  
`mkstring (0)` gives a `Bit_string` of one element, i.e. 0
- ✎ `length` :

The number of Bits in the `Bit_string` passed as parameter.  
`length (bitstr('0110')) = 4`
- ✎ `first` :

The value of the first Bit in the `Bit_string` passed as parameter. If the length of the `Bit_string` is 0, then it is an error to call the first operator.  
`first (bitstr ('10')) = 1`
- ✎ `last` :

The value of the last Bit in the `Bit_string` passed as parameter. If the length of the `Bit_string` is 0, then it is an error to call the last operator.  
`last (bitstr ('10')) = 0`
- ✎ `// (concatenation)` :

The result is a `Bit_string` with all the elements in the first parameter, followed by all the elements in the second parameter.  
`bitstr('01')//bitstr('10') = bitstr('0110')`
- ✎ `substring` :

The result is a copy of a part of the `Bit_string` passed as first parameter. The copy starts at the index given as second parameter. The first Bit has index 0. The length of the copy is specified by the third parameter. It is an error to try to access elements outside of the true

length of the first parameter.

```
substring (bitstr('0110'), 1, 2) = Bitstr('11')
```

≈ bitstr :

This Telelogic-specific operator converts a charstring containing only characters 0 and 1, to a Bit\_string with the same length and with the Bit elements set to the corresponding values.

≈ hexstr :

This Telelogic-specific operator converts a charstring containing HEX values (0-9, A-F, a-f) to a Bit\_string. Each HEX value is converted to four Bit elements in the Bit\_string.

```
hexstr('a') = bitstr('1010'),
hexstr('8f') = bitstr('10001111')
```

≈ not :

The result is a Bit\_string with the same length as the parameter, where the not operator in the Bit sort has been applied to each element, that is each element has been inverted.

```
not bitstr ('0110') = bitstr ('1001')
```

≈ and :

The result is a Bit\_string with the same length as the longest of the two parameters. Each Bit element in the result is computed by applying the and operator in Bit sort to the corresponding Bit elements in the parameters. This calculation is performed up to the length of the shortest parameter. The remaining Bits in the result (if any) are set to 0.

```
bitstr('01101') and bitstr('101') = bitstr('00100')
```

≈ or :

The result is a Bit\_string with the same length as the longest of the two parameters. Each Bit element in the result is computed by applying the or operator in Bit sort to the corresponding Bit elements in the parameters. This calculation is performed up to the length of the shortest parameter. The remaining Bits in the result (if any) are set to 1.

```
bitstr('0110') or bitstr('00110') = bitstr('01111')
```

≈ xor :

The result is a Bit\_string with the same length as the longest of the two parameters. Each Bit element in the result is computed by applying the xor operator in Bit sort to the corresponding Bit elements in the parameters. This calculation is performed up to the length of the shortest parameter. The remaining Bits in the result (if any) are set to 1.

```
bitstr('10100') xor bitstr('1001') = bitstr('00111')
```

≈ => (implication) :

The result is a Bit\_string with the same length as the longest of the two parameters. Each Bit element in the result is computed by applying the => operator in Bit sort to the corresponding Bit elements in the parameters. This calculation is performed up to the length of the shortest parameter. The remaining Bits in the result (if any) are set to 1.

```
bitstr ('1100') => bitstr ('0101') = bitstr ('0111')
```

It is also possible to access Bit elements in a Bit\_string by indexing a Bit\_string variable. Assume that B is a Bit\_string variable. Then it is possible to write:

```
task B(2) := B(3);
```

This would mean that Bit number 2 is assigned the value of Bit number 3 in the variable B. It is an error to index a Bit\_string outside of its length.

Note:

The first Bit in a Bit\_string has index 0, whereas most other string types in SDL start with index 1!

## Boolean

The newtype `Boolean` can only take two values, `false` and `true`. The operators that are available for Boolean values are:

```
"not" : Boolean -> Boolean
"and" : Boolean, Boolean -> Boolean
"or"  : Boolean, Boolean -> Boolean
"xor" : Boolean, Boolean -> Boolean
"=>" : Boolean, Boolean -> Boolean
```

These operators are defined according to the following:

≍ `not` :  
inverts the value.

```
not false = true
not true  = false
```

≍ `and` :  
If both parameters are true then the result is true, else it is false.

```
false and false = false
false and true  = false
true  and false = false
true  and true  = true
```

≍ `or` :  
If both parameters are false then the result is false, else it is true.

```
false or false = false
false or true  = true
true  or false = true
true  or true  = true
```

≍ `xor` :  
If parameters are different then the result is true, else it is false.

```
false xor false = false
false xor true  = true
true  xor false = true
true  xor true  = false
```

≍ `=>` (implication) :  
If the first parameter is true and second is false then the result is false, else it is true.

```
false => false = true
false => true  = true
true  => false = false
true  => true  = true
```

The Bit sort, discussed above, has most of its properties in common with the Boolean sort. By

replacing 0 with `false` and 1 with `true` the sorts are identical. Normally it is recommended to use Boolean instead of Bit; for a more detailed discussion see "[Bit](#)".

## Character

The `character` sort is used to represent the ASCII characters. The printable characters have literals according to the following example:

```
'a' '-' '?' '2' 'P' ''
```

Note that the character ' ' is written twice in the literal. For the non-printable characters, specific literal names have been included in the Character sort. The following:

```
NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
BS, HT, LF, VT, FF, CR, SO, SI,
DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
CAN, EM, SUB, ESC, IS4, IS3, IS2, IS1
```

correspond to the characters with number 0 to 31, while the literal

```
DEL
```

corresponds to the character number 127.

The operators available in the Character sort are:

```
"<" : Character, Character -> Boolean;
"<=" : Character, Character -> Boolean;
">" : Character, Character -> Boolean;
">=" : Character, Character -> Boolean;
num : Character -> Integer;
chr : Integer -> Character;
```

The interpretation of these operators are:

≪ <, <=, >, >= :

These relation operators work with the character numbers according to the ASCII table.

≪ num :

This operator converts a Character value to its corresponding character number. For

example: `num('A') = 65`

≪ chr :

This operator converts an Integer value to its corresponding character. If the parameter is less than 0 or bigger than 255, it is first taken modulo 256 (using the mod operator in sort Integer). For example:

`chr(65) = 'A'`

In Z.100 characters in the range 0 to 127 are supported. However Telelogic has introduced support for characters in the range 0 to 255. This means two things

The operator `num` works modulo 256, not modulo 128 as it is defined in Z.100.

The following literals (128 to 255) are added to the Character sort:

```

E_NUL, E_SOH, E_STX, E_ETX, E_EOT, E_ENQ, E_ACK,
E_BEL,
E_BS, E_HT, E_LF, E_VT, E_FF, E_CR, E_SO, E_SI,
E_DLE, E_DC1, E_DC2, E_DC3, E_DC4, E_NAK, E_SYN,
E_ETB,
E_CAN, E_EM, E_SUB, E_ESC, E_IS4, E_IS3, E_IS2,
E_IS1,
' ', '!', '¢', '£', '¤', '¥', '¦', '§', '¨',
'©', 'ª', «, ¬, '\x7f', '®', '¯',
'°', '±', '²', '³', ´, 'µ', ¶, '·',
'¸', '¹', 'º', »', '¼', '½', '¾', '¿',
'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç',
'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î', 'Ï',
'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×',
'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß',
'à', 'á', 'â', 'ã', 'ä', 'å', 'æ', 'ç',
'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï',
'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷',
'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ';

```

## Charstring

The `Charstring` sort is used to represent strings or sequences of characters. There is no limit for the length of a `Charstring` value. `Charstring` literals are written as a sequence of characters enclosed between two single quotes: `'abc'`. If the `Charstring` should contain a quote (') it must be written twice.

```

'abcdef0123'
'$%@^&'
'1"2"3' /* denotes 1'2'3 */
' /* empty Charstring */

```

The following operators are available for `Charstrings`:

```

mkstring : Character -> Charstring;
length : Charstring -> Integer;
first : Charstring -> Character;
last : Charstring -> Character;
"/" : Charstring, Charstring -> Charstring;
substring : Charstring, Integer, Integer
-> Charstring;

```

These operators are defined as follows:

- ✎ `mkstring` :  
This operator takes one `Character` value and converts it to a `Charstring` of length 1. For example: if `c` is a variable of type `Character`, then `mkstring(c)` is a `Charstring` containing character `c`.
- ✎ `length` :  
This operator takes a `Charstring` as parameter and returns its number of characters.  
`length ('hello') = 5`
- ✎ `first` :  
The value of the first `Character` in the `Charstring` passed as parameter. If the length of the

Charstring is 0, then it is an error to call the first operator.

```
first ('hello') = 'h'
```

⌘ last :

The value of the last Character in the Charstring passed as parameter. If the length of the Charstring is 0, then it is an error to call the last operator.

```
last ('hello') = 'o'
```

⌘ // (concatenation) :

The result is a Charstring with all the elements in the first parameter, followed by all the elements in the second parameter.

```
'he' // 'llo' = 'hello'.
```

⌘ substring :

The result is a copy of a part of the Charstring passed as first parameter. The copy starts at the index given as second parameter (Note: first Character has index 1). The length of the copy is specified by the third parameter. It is an error to try to access elements outside of the true length of the first parameter.

```
substring ('hello', 3, 2) = 'll'
```

It is also possible to access Character elements in a Charstring by indexing a Charstring variable. Assume that *c* is a Charstring variable. Then it is possible to write:

```
task C(2) := C(3);
```

This would mean that Character number 2 is assigned the value of Character number 3 in the variable *c*.

Note:

The first Character in a Charstring has index 1.

### IA5String, NumericString, PrintableString, VisibleString

These Z.105 specific character string types are all syntypes of Charstring with restrictions on the allowed Characters that may be contained in a value. These sorts are mainly used as a counterpart of the ASN.1 types with the same names. The restrictions are:

⌘ IA5String :

only NUL:DEL, i.e only characters in the range 0 to 127.

⌘ NumericString :

only '0':'9' and ' '

⌘ PrintableString :

only 'A':'Z', 'a':'z', '0':'9', ' ', '(':')', '+':'/', ':', '=', '?'

⌘ VisibleString:

only ' ':'~'

It is recommended to use these types only in relation with ASN.1 or TTCN. In other cases use Charstring.

### Duration, Time

The Time and Duration sorts have their major application area in connection with timers. The first parameter in a Set statement is the time when the timer should expire. This value should be of sort Time.

Both Time and Duration have literals with the same syntax as real values. Example:

245.72 0.0032 43

The following operators are available in the Duration sort:

"+" : Duration, Duration -> Duration;  
 "-" : Duration -> Duration;  
 "-" : Duration, Duration -> Duration;  
 "\*" : Duration, Real -> Duration;  
 "\*" : Real, Duration -> Duration;  
 "/" : Duration, Real -> Duration;  
 ">" : Duration, Duration -> Boolean;  
 "<" : Duration, Duration -> Boolean;  
 ">=" : Duration, Duration -> Boolean;  
 "<=" : Duration, Duration -> Boolean;

The following operators are available in the Time sort:

"+" : Time, Duration -> Time;  
 "+" : Duration, Time -> Time;  
 "-" : Time, Duration -> Time;  
 "-" : Time, Time -> Duration;  
 "<" : Time, Time -> Boolean;  
 "<=" : Time, Time -> Boolean;  
 ">" : Time, Time -> Boolean;  
 ">=" : Time, Time -> Boolean;

The interpretation of these operators are rather straightforward, as they correspond directly to the ordinary mathematical operators for real numbers. There is one "operator" in SDL that returns a Time value; `now` which returns the current global system time.

Time should be used to denote "a point in time", while Duration should be used to denote a "time interval". SDL does not specify what the unit of time is. In the SDL suite, the time unit is usually 1 second.

Example 3 : Timers in SDL

SET (now + 2.5, MyTimer)

After the above statement, SDL timer `MyTimer` will expire after 2.5 time units (usually seconds) from now.

You should note that according to SDL, Time and Duration (and Real) possess the true mathematical properties of real numbers. In an implementation, however, there are of course limits on the range and precision of these values.

### Integer, Natural

The `Integer` sort in SDL is used to represent the mathematical integers. `Natural` is a syntype of `Integer`, allowing only integers greater than or equal to zero.

Integer literals are defined using the ordinary integer syntax. Example:

```
0 5 173 1000000
```

Negative integers are obtained by using the unary - operator given below. The following operators are defined in the Integer sort:

```
"-" : Integer -> Integer;
"+" : Integer, Integer -> Integer;
"-" : Integer, Integer -> Integer;
"*" : Integer, Integer -> Integer;
"/" : Integer, Integer -> Integer;
"mod" : Integer, Integer -> Integer;
"rem" : Integer, Integer -> Integer;
"<" : Integer, Integer -> Boolean;
">" : Integer, Integer -> Boolean;
"<=" : Integer, Integer -> Boolean;
">=" : Integer, Integer -> Boolean;
float : Integer -> Real;
fix : Real -> Integer;
```

The interpretation of these operators are given below:

- ⌘ - (unary, i.e. one parameter) :  
Negate a value, e.g. -5.
- ⌘ +, -, \* :  
These operators correspond directly to their mathematical counterparts.
- ⌘ / :  
Integer division, e.g.  $10/5 = 2$ ,  $14/5 = 2$ ,  $-8/5 = -1$
- ⌘ mod, rem :  
modulus and remainder at integer division. mod always returns a positive value, while rem may return negative values, e.g.  
 $14 \text{ mod } 5 = 4$ ,  $14 \text{ rem } 5 = 4$ ,  $-14 \text{ mod } 5 = 1$ ,  $-14 \text{ rem } 5 = -4$
- ⌘ <, <=, >, >= :  
These operators correspond directly to their mathematical counterparts.
- ⌘ float :  
This operator converts an integer value to the corresponding Real number, for example:  
float (3) = 3.0
- ⌘ fix :  
This operator converts a real value to the corresponding Integer number. It is performed by removing the decimal part of the Real value.  
fix(3.65) = 3, fix(-3.65) = -3

## NULL

NULL is a sort coming from ASN.1, defined in Z.105. NULL does occur rather frequently in older protocols specified with ASN.1. ASN.1 has later been extended with better alternatives, so NULL should normally not be used. The sort NULL only contains one value, NULL.

## Object\_identifier

The Z.105-specific sort `Object_identifier` also comes from ASN.1. Object identifiers usually

identify some globally well-known definition, for example a protocol, or an encoding algorithm. Object identifiers are often used in open-ended applications, for example in a protocol where one party could say to the other "I support protocol version X". "Protocol version X" could be identified by means of an object identifier.

An `Object_identifier` value is a sequence of Natural values. This sort contains one literal, `emptystring`, that is used to represent an `Object_identifier` with length 0. The operators defined in this sort are:

```
mkstring : Natural -> Object_identifier
length : Object_identifier -> Integer
first : Object_identifier -> Natural
last : Object_identifier -> Natural
"/" : Object_identifier, Object_identifier
-> Object_identifier
substring : Object_identifier, Integer, Integer
-> Object_identifier
append : in/out Object_identifier, Natural;
(. .) : * Natural -> Object_identifier
```

These operators are defined as follows:

✎ `mkstring` :

This operator takes one Natural value and converts it to an `Object_identifier` of length 1. `mkstring (8)` gives an `Object_identifier` consisting of one element, i.e. 8.

✎ `length` :

This operator takes an `Object_identifier` as parameter and returns its number of object elements, i.e. Natural values.

```
length (mkstring (8)//mkstring(6)) = 2
length (emptystring) = 0
```

✎ `first` :

The value of the first Natural in the `Object_identifier` passed as parameter. If the length of the `Object_identifier` is 0, then it is an error to call the first operator.

```
first (mkstring (8)//mkstring(6)) = 8
```

✎ `last` :

The value of the last Natural in the `Object_identifier` passed as parameter. If the length of the `Object_identifier` is 0, then it is an error to call the last operator.

```
last (mkstring (8)//mkstring(6)) = 6
```

✎ `// (concatenation)` :

The result is a `Object_identifier` with all the elements in the first parameter, followed by all the elements in the second parameter.

`mkstring (8) // mkstring (6)` gives an `Object_identifier` of two elements, 8 followed by 6.

✎ `substring` :

The result is a copy of a part of the `Object_identifier` passed as first parameter. The copy starts at the index given as second parameter (Note: first Natural has index 1). The length of the copy is specified by the third parameter. It is an error to try to access elements outside of the true length of the first parameter.

```
substring (mkstring (8) //mkstring(6), 2, 1) =mkstring(6)
```

✎ `append` :

`append` is a Telelogic extension and can be used to add a new component to the end of an existing `Object_identifier`. `append` takes a variable as first parameter and a Natural value as second. The variable is then updated to include the second parameter as last component in

the `Object_identifier`. The reason for introducing this operator is that:

```
task append(V, 12);
```

is much more efficient than performing the same calculation as

```
task V := V // mkstring(12);
```

≠ `(. .)`:

The `(. .)` expression, which is a Telelogic extension, is an application of the implicit make operator. The make operator takes a sequence of Natural values and returns an `Object_identifier` that contains these value in the order they are given.

`Obj_id_var := (. 1, 2, 3 .)` would give an `Object_identifier` containing 1, 2 and 3.

It is also possible to access the Natural elements in an `Object_identifier` by indexing an `Object_identifier` variable. Assume that `c` is a `Object_identifier` variable. Then it is possible to write:

```
task C(2) := C(3);
```

This would mean that the Natural at index 2 is assigned the value of the Natural at index 3 in the variable `c`. Note that the first Natural in an `Object_identifier` has index 1. It is an error to index an `Object_identifier` outside of its length.

## Octet

The Z.105-specific sort `Octet` is used to represent eight-bit values, i.e. values between 0 and 255. In C this would correspond to unsigned char. There are no explicit literals for the Octet sort. Values can, however, easily be constructed using the conversion operators `i2o` and `o2i` discussed below.

The following operators are defined in Octet:

```
"not" : Octet -> Octet;
"and" : Octet, Octet -> Octet;
"or" : Octet, Octet -> Octet;
"xor" : Octet, Octet -> Octet;
"==" : Octet, Octet -> Octet;
"<" : Octet, Octet -> Boolean;
"<=" : Octet, Octet -> Boolean;
">" : Octet, Octet -> Boolean;
">=" : Octet, Octet -> Boolean;
shifl : Octet, Integer -> Octet;
shiftr : Octet, Integer -> Octet;
"+" : Octet, Octet -> Octet;
"-" : Octet, Octet -> Octet;
"*" : Octet, Octet -> Octet;
"/" : Octet, Octet -> Octet;
"mod" : Octet, Octet -> Octet;
"rem" : Octet, Octet -> Octet;
i2o : Integer -> Octet;
o2i : Octet -> Integer;
bitstr : Charstring -> Octet;
hexstr : Charstring -> Octet;
```

The interpretation of these operators is as follows:

⌘ not, and, or, xor, => :

Apply the corresponding Bit operator for each of the eight bits in the Octet. For example:

```
not bitstr ('00110101') = bitstr ('11001010')
```

⌘ <, <=, >, >= :

Ordinary relation operators for the Octet values.

⌘ shiftl, shiftr :

These Telelogic-specific operators are defined as left and right shift in C, so `shiftl(a,b)` is defined as `a<<b` in C.

```
shiftl (bitstr('1'), 4) = bitstr('10000')
```

```
shiftr (bitstr('1010'), 2) = bitstr ('10')
```

⌘ +, -, \*, /, mod, rem :

These operators are the mathematical corresponding operators. All operations are, however, performed modulus 256.

```
i2o(250) + i2o(10) = i2o(4), o2i(i2o(4)-i2o(6)) = 254
```

⌘ i2o :

This Telelogic-specific operator converts an Integer value to the corresponding Octet value.

```
i2o (128) = hexstr ('80')
```

⌘ o2i :

This Telelogic-specific operator converts an Octet value to the corresponding Integer value.

```
o2i (hexstr ('80')) = 128
```

⌘ bitstr :

This Telelogic-specific operator converts a charstring containing eight Bit values ("0" and "1") to an Octet value.

```
bitstr('00000011') = i2o(3)
```

⌘ hexstr :

This Telelogic-specific operator converts a charstring containing two HEX values ("0"- "9", "a"- "f", "A"- "F") to an Octet value.

```
hexstr('01') = i2o(1), hexstr('ff') = i2o(255)
```

It is also possible to read the individual bits in an Octet value by indexing an Octet variable. The index should be in the range 0 to 7.

## Octet\_string

The Z.105-specific sort `Octet_string` represents a sequence of `Octet` values. There is no limit on the length of the sequence. The operators defined in the `Octet_string` sort are:

```
mkstring : Octet -> Octet_string;
length : Octet_string -> Integer;
first : Octet_string -> Octet;
last : Octet_string -> Octet;
"/" : Octet_string, Octet_string
-> Octet_string;
substring : Octet_string, Integer, Integer
-> Octet_string;
bitstr : Charstring -> Octet_string;
hexstr : Charstring -> Octet_string;
bit_string : Octet_string -> Bit_string;
octet_string : Bit_string -> Octet_string;
```

These operators are defined as follows:

✎ `mkstring` :

This operator takes an Octet value and converts it to a Octet\_string of length 1.

`mkstring (i2o(10))` gives an Octet\_string containing one element.

✎ `length` :

The number of Octets in the Octet\_string passed as parameter.

`length (i2o (8)//i2o (6)) = 2`

`length ( hexstr ('0f3d88')) = 3`

`length ( bitstr ('')) = 0`

✎ `first` :

The value of the first Octet in the Octet\_string passed as parameter. If the length of the Octet\_string is 0, then it is an error to call the first operator.

`first ( hexstr ('0f3d88')) = hexstr('0f') (= i2o(15))`

✎ `last` :

The value of the last Octet in the Octet\_string passed as parameter. If the length of the Octet\_string is 0, then it is an error to call the last operator.

`last ( hexstr ('0f3d88')) = hexstr('88') (= i2o(136))`

✎ `// (concatenation)` :

The result is an Octet\_string with all the elements in the first parameter, followed by all the elements in the second parameter.

`hexstr('0f3d')//hexstr('884F') = hexstr('0f3d884f')`

✎ `substring` :

The result is a copy of a part of the Octet\_string passed as first parameter. The copy starts at the index given as the second parameter. The length of the copy is specified by the third parameter. It is an error to try to access elements outside of the true length of the first parameter.

`substring(hexstr('0f3d889C'), 3, 2) = hexstr('889c')`

✎ `bitstr` :

This Telelogic-specific operator converts a charstring containing only characters 0 and 1, to an Octet\_string with an appropriate length and with the Octet elements set to the value defined in the sequences of eight bits. If the Charstring length is not a multiple of eight, it is padded with zeros.

`bitstr ('101') = bitstr ('10100000')`

✎ `hexstr` :

This Telelogic-specific operator converts a charstring containing HEX values (0-9, A-F, a-f) to an Octet\_string. Each pair of HEX values are converted to one Octet element in the Octet\_string. If the Charstring length is not a multiple of two, it is padded with a zero.

`hexstr ('f') = hexstr ('f0')`

✎ `bit_string and octet_string` :

These two operators convert values between Bit\_string and Octet\_string.

It is also possible to access the Octet elements in an Octet\_string by indexing an Octet\_string variable. Assume that C is an Octet\_string variable. Then it is possible to write:

```
task C(2) := C(3);
```

This would mean that the Octet at index 2 is assigned the value of Octet at index 3 in the variable C. It is an error to index an Octet\_string outside of its length.

Note:

The first Octet in an Octet\_string has index 1.

**Pid**

The sort `Pid` is used as a reference to process instances. `Pid` has only one literal, `Null`. All other values are obtained from the SDL predefined variables `Self`, `Sender`, `Parent`, and `Offspring`.

**Real**

`Real` is used to represent the mathematical real values. In an implementation there are of course always restrictions in size and precision of such values. Examples of `Real` literals:

2.354 0.9834 23 100023.001

The operators defined in the `Real` sort are:

```
"-" : Real -> Real;
"+" : Real, Real -> Real;
"-" : Real, Real -> Real;
"*" : Real, Real -> Real;
"/" : Real, Real -> Real;
"<" : Real, Real -> Boolean;
">" : Real, Real -> Boolean;
"<=" : Real, Real -> Boolean;
">=" : Real, Real -> Boolean;
```

All these operators have their ordinary mathematical meaning.

**User Defined Sorts**

All the predefined sorts and syntypes discussed in the previous section can be directly used in, for example, variable declarations. In many circumstances it is however suitable to introduce new sorts and syntypes into a system to describe certain properties of the system. A user-defined sort or syntype can be used in the unit where it is defined, and also in all its subunits.

**Syntypes**

A *syntype* definition introduces a new type name which is fully compatible with the base type. This means that a variable of the syntype may be used in any position where a variable of the base type may be used. The only difference is the range check in the syntype. One exception exists. The actual parameter that corresponds to a formal in/out parameter must be of the same syntype as the formal parameter. Otherwise proper range tests cannot be performed.

Syntypes are useful for:

- ⌘ Introducing a new name for an existing type
- ⌘ Introducing a new type that has the same properties as an existing type, but with a restricted value range
- ⌘ Defining index sorts used in arrays

Example 4 : Syntype definition

syntype smallint = integer

```

constants 0:10
endsyntype;

```

In this example `smallint` is the new type name, `integer` is the base type, and `0:10` is the range condition. Range conditions can be more complex than the one above. It may consist of a list of conditions, where each condition can be (assume `x` to be a suitable value):

- ≠ `=x` a single value `x` is allowed
- ≠ `x` same as `=x`
- ≠ `/=x` all values except `x` are allowed
- ≠ `>x` all values `>x` are allowed
- ≠ `>=x` all values `>=x` are allowed
- ≠ `<x` all values `<x` are allowed
- ≠ `<=x` all values `<=x` are allowed
- ≠ `x:y` all values `>=x` and `<=y` are allowed

#### Example 5 : Syntype definition

```

syntype strangeint = integer
constants <-5, 0:3, 5, 8, >=13
endsyntype;

```

In this example all values `<-5, 0, 1, 2, 3, 5, 8, >=13` are allowed.

The range check introduced in a syntype is tested in the following cases (assuming that the variable, signal parameter, formal parameter involved is defined as a syntype):

- ≠ Assignment to a variable
- ≠ Assigning a value to a signal parameter in an output (also for the implicit signals used in connection with import and remote procedure calls)
- ≠ Assigning a value to an `IN` parameter in a procedure call
- ≠ Assigning a value to a process parameter in a create request action
- ≠ Assigning a value to a variable in an input
- ≠ Assigning a value to an operator parameter (also for the operator result)
- ≠ Assigning a value to a timer parameter in set, reset, or active

### Enumeration Sorts

An *enumeration sort* is a sort containing only the values enumerated in the sort. If some property of the system can take a relatively small number of distinct values and each value has a name, an enumeration sort is probably suitable to describe this property. Assume for example a key with three positions; off, stand-by, and service-mode. A suitable sort to describe this would be:

#### Example 6 : Enumeration sort

```

newtype KeyPosition
literals Off, Stand_by, Service_mode
endnewtype;

```

A variable of sort `KeyPosition` can take any of the three values in the literal list, but no other.

### Struct

The *struct* concept in SDL can be used to make an aggregate of data that belongs together. Similar features can be found in most programming languages. In C, for example, it is also called struct, while in Pascal it is the record concept that has these properties. If, for example, we would like to describe a person and would like to give him a number of properties or attributes, such as name, address, and phone number, we can write:

```
newtype Person struct
  Name Charstring;
  Address Charstring;
  PhoneNumber Charstring;
endnewtype;
```

A struct contains a number of components, each with a name and a type. If we now define variables of this struct type,

```
dcl p1, p2 Person;
```

it is possible to work directly with complete struct values, like in assignments, or in tests for equality. Also individual components in the struct variable can be selected or changed.

```
task p1 := (. 'Peter', 'Main Road, Smalltown',
'+46 40 174700' .);
task BoolVar := p1 = p2;
task p2 ! Name := 'John';
task CharstringVar := p2 ! Name;
```

The first task is an assignment on the struct level. The right hand side, i.e. the ( . ) expression, is an application of the implicit make operator, that is present in all structs. The make operator takes a value of the first component sort, followed by a value of the second component sort, and so on, and returns a struct value where the components are given the corresponding values. In the example above, the component *Name* in variable *p1* is given the value 'Peter'. The second task shows a test for equality between two struct expressions. The third and fourth task shows how to access a component in a struct. A component is selected by writing:

```
VariableName ! ComponentName
```

Such component selection can be performed both in a expression (then usually called *extract*) and in the left hand side of an assignment (then usually called *modify*).

## Bit Fields

A *bit field* defines the size in bits for a struct component. This feature is not part of the SDL Recommendation, but rather introduced by Telelogic to enable the generation of C bit fields from SDL. This means that the syntax and semantics of bit fields follow the C counterpart very much.

### Example 7 : Bit fields

```
newtype example struct
  a Integer : 4;
  b UnsignedInt : 2;
  c UnsignedInt : 1;
  : 0;
```

```

d Integer : 4;
e Integer;
endnewtype;

```

The following rules apply to bit fields:

- ⚡ The meaning of the bit field size, i.e. the `: x` (where `x` is an integer number) is the same as in C. When generating C code from SDL, the `: x` is just copied to the C struct that is generated from the SDL struct.
- ⚡ `: 0` in SDL is translated to `int : 0` in C.
- ⚡ As C only allows `int` and `unsigned int` for bit field components the same rule is valid in SDL: only `Integer` and `UnsignedInt` (from package `ctypes`) may be used.

Bit fields should only be used when it is necessary to generate C bit fields from SDL. Bit fields should not be used as an alternative to syntypes with a constants clause; the SDL suite does not check violations to the size of the bit fields.

### Optional and Default values

To simplify the translation of ASN.1 types to SDL sorts, two new features have been introduced into structs. Struct components can be *optional* and they can have *default values*. Note that these features have their major application area in connection with ASN.1 data types and applying them in other situations is probably not a good idea, as they are not standard SDL-96.

Example 8 : Optional and default values

```

newtype example struct
a Integer optional;
b Charstring;
c Boolean := true;
d Integer := 4;
e Integer optional;
endnewtype;

```

The default values for component `c` and `d`, means that these components are initialized to the given values.

An *optional* component may or may not be present in a struct value. Initially an optional component is not present. It becomes present when it is assigned a value. It is an error to access a component that is not present. It is possible to test if an optional component is present or not by calling an implicit operator called

*ComponentNamepresent*

In the example above `apresent(v)` and `epresent(v)` can be used to test whether components `a` and `e` are present or not, in the value stored in variable `v`. A component that is present can be set to absent, i.e. not present, again by calling the implicit operator

*ComponentNameabsent*

In the example above `aabsent(v)` and `eabsent(v)` can be used to set the components to absent. Note that the absent operators are operators without result.

Components with default values also have `present` and `absent` operators in the same way as optional components. They however do not have the same semantics as for optional components. A component with default value always has a value! Present and absent instead have to do with encoding and decoding of ASN.1 values. A component that contains its default value, i.e. is absent, is in some encoding schemes not encoded.

A component with default value is initialized with the default value and has `present` equal to false. Present can for components with default values be seen as "is explicitly assigned some value". This means that when a component with default value is assigned a value, in an assignment for example, `present` will become true (even if the component is assigned the default value). The `absent` operator can be used to set the component back to absent. This means that the `absent` operator performs two things: assigns the component the default value and sets `present` to false.

According to Z.105, the `make` operator for a struct does not include components that are optional or contain a default value. Optional components always become absent and components with default values are always initialized with their default values. The `struct example` in the previous example only contains one component that is not optional and does not contain a default value. This means that a variable `v` of this type can be assigned a struct value by:

```
task v := (. 'hello' .);
```

If we want to set the other components, this have to be performed in a sequence of assignments after this assignment.

To simplify assigning a complete struct value to a struct in these cases, Telelogic provide an alternative interpretation of `make` for a struct. You specify that you want to use this alternative interpretation of `make` by selecting *Generate > Analyze > Details > Semantic Analysis > Include optional fields in make operator*.

The alternative `make` always takes all components as parameters. By inserting an empty position you can specify that you want the component not present or given its default value. By giving a value you specify the value to be assigned to that component. Using the example above again it is possible to write:

```
task v := (. 1, 'hello', , 10, .);
```

This means that the first, second, and fourth components are given explicit values, while the third and fifth becomes absent.

## Choice

The new concept *choice* is introduced into SDL as a means to represent the ASN.1 concept CHOICE. This concept can also be very useful while developing pure SDL data types. The choice in SDL can be seen as a C *union* with an implicit tag field.

Example 9 : Choice

```
newtype C1 choice
a Integer;
b Charstring;
c Boolean;
endnewtype;
```

The example above shows a choice with three components. The interpretation is that a variable of a choice type can only contain one of the components at a time, so in the example above a value of `c1` either contains an Integer value, a Charstring value, or a Boolean value.

Example 10 : Working with a choice type

```
DCL var C1, charstr Charstring;

TASK var := a : 5; /* assign component a */
TASK var!b := 'hello'; /* assign component b
(a becomes absent) */
TASK charstr := var!b; /* get component b */
```

The above example shows how to modify and extract components of a choice type. In this respect, choice types are identical to struct types, except the `a : 5` notation to denote choice values, whereas struct values are described using `(. ... .)`.

Extracting a component of a choice type that is not present results in a run-time error. Therefore it is necessary to be able to determine which component is active in a particular value. For that purpose there are a number of implicit operators defined for a choice.

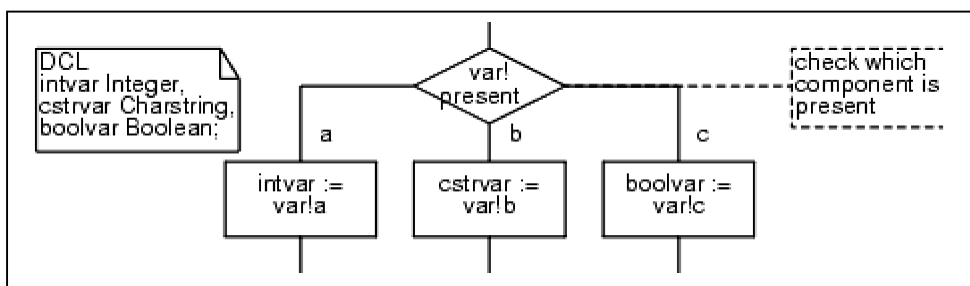
```
var!present
```

where `var` is a variable of a choice type, returns a value which is the name of the active component. This is made possible by introducing an implicit enumeration type with literals with the same names as the choice components. Note that this enumeration type is implicit and should not be inserted by you. Given the example above, it is allowed to test:

```
var!present = b
```

This is illustrated in [Figure 26](#).

Figure 26 : Check which component of a choice is present



It is also possible to test if a certain component is active or not, by using the implicit boolean operators `ComponentNamepresent`. To check if component `b` in the example above is present it is thus possible to write:

```
bpresent(v)
```

The information about which component that is active can be accessed using the `present` operators, but it is not possible to change it. This information is automatically updated when a component in a choice variable is assigned a value.

The purpose of choice is to save memory or bandwidth. As it is known that only one component at a time can contain a value, the compiler can use overlay techniques to reduce the total memory for the type. Also sending a choice value over a physical connection saves time, compared to sending a corresponding struct.

The choice construct is Telelogic-specific, and not part of recommendation Z.105, so if you want to write portable SDL, you should not use choice. Choice replaces the SDL suite `#UNION` code generator directive. It is recommended to replace `#UNION` directives by choice, as the SDL suite has better tool support for the latter.

## Inherits

It is possible to create a new sort by *inheriting* information from another sort. It is possible to specify which operators and literals that should be inherited and it is then possible to add new operators and literals in the new type.

Note that it is not really possible to change the type in itself by using inheritance. It is, for example, not possible to add a new component to a struct when the struct is inherited.

Our experience with inheritance so far has been that it is not as useful as it might seem in the beginning, and that sometimes the use of inheritance leads to the need of qualifiers in a lot of places, as many expressions are no longer semantically valid.

### Example 11 : Inherits

```
newtype NewInteger inherits Integer
operators all;
endnewtype;
```

In the example above a new type `NewInteger` is introduced. This type is distinct from `Integer`, i.e. an `Integer` expression or variable is not allowed where a `NewInteger` is expected, and a `NewInteger` expression or variable is not allowed where an `Integer` is expected. Since in the example all literals and operators are inherited, all the integer literals 0, 1, 2, ..., are also available as `NewInteger` literals. For operators it means that all operators having `Integer` as parameter or result type are copied, with the `Integer` parameter replaced with a `NewInteger` parameter. This is true for all operators, not only those defined in the `Integer` sort, which may give unexpected effects, which will be illustrated below.

### Example 12 : Inherited operators

The following operators are some of the operators having `Integer` as parameter or result type:

```
"+" : Integer, Integer -> Integer;
"-" : Integer -> Integer;
"mod" : Integer, Integer -> Integer;
length : Charstring -> Integer;
```

The type `NewInteger` defined above will inherit these and all the others having integer as parameter or result type. Note that `length` is defined in the `Charstring` sort.

```
"+" : NewInteger, NewInteger -> NewInteger;
"-" : NewInteger -> NewInteger;
```

```
"mod" : NewInteger, NewInteger -> NewInteger;
length : Charstring -> NewInteger;
```

With this `NewInteger` declaration, statements like

```
decision length(Charstring_Var) > 5;
```

are no longer correct in the SDL system. It is no longer possible to determine the types in the expression above. It can either be the length returning integer that is tested against an integer literal, or the length returning a `NewInteger` value that is tested against a `NewInteger` literal.

It is possible to avoid this kind of problem by specifying explicitly the operators that should be inherited.

Example 13 : Inherits

```
newtype NewInteger inherits Integer
operators ("+", "-", "*", "/")
endnewtype;
```

Now only the enumerated operators are inherited and the problem with `length` that was discussed above will not occur.

## Predefined Generators

### Array

The predefined generator `Array` takes two generator parameters, an index sort and a component sort. There are no restrictions in SDL on the index and component sort.

Example 14 : Array instantiation

```
newtype A1 Array(Character, Integer)
endnewtype;
```

The example above shows an instantiation of the `Array` generator with `Character` as index sort and `Integer` as component sort. This means that we now have created a data structure that contains one `Integer` value for each possible `Character` value. To obtain the component value connected to a certain index value it is possible to index the array.

Example 15 : Using an array type

```
dcl Var_A1 A1; /* Assume sort in example above */

task Var_A1 := (. 3.);
task Var_Integer := Var_A1('a');
task Var_A1('x') := 11;

decision Var_A1 = (. 11.);
(true) : ...
...
enddecision;
```

The example above shows how to work with arrays. First we have the expression `(. 3 .)`. This is an application of the *make!* operator defined in all array instantiations. The purpose is to return an array value with all components set to the value specified in *make*. The first task above thus assigns the value 3 to all array components. Note that this is an assignment of a complete array value.

In the second task the value of the array component at index 'a' is extracted and assigned to the integer variable `Var_Integer`. In the third task the value of the array component at index 'x' is modified and given the new value 11. The second and third task show applications of the operators *extract!* and *modify!* which are present in all array instantiations. Note that the operators *extract!*, *modify!*, and *make!* can only be used in the way shown in the example above. It is not allowed to directly use the name of these operators.

In the last statement, the decision, an equal test for two array values is performed. Equal and not equal are, as well as assignment, defined for all sorts in SDL.

The typical usage of arrays is to define a fixed number of elements of the same sort. Often a syntype of Integer is used for the index sort, as in the following example, where an array of 11 Pids is defined with indices 0 to 10.

Example 16 : Typical array definition

```
syntype indexsort = Integer
constants 0:10
endsyntype;

newtype PidArray Array (indexsort, Pid)
endnewtype;
```

Unlike most ordinary programming languages, there are no restrictions on the index sort in SDL. In most programming languages the index type must define a finite range of values possible to enumerate. In C, for example, the size of an array is specified as an integer constant, and the indices in the array range from 0 to the (size-1). In SDL, however, there are no such limits.

Example 17 : Array with infinite number of elements.

```
newtype RealArr Array (Real, Real)
endnewtype;
```

Having Real as index type means that there is an infinite number of elements in the array above. It has, however, the same properties as all other arrays discussed above. This kind of more advanced arrays sometimes can be a very powerful concept that can be used for implementing, for example, a mapping table between different entities.

Example 18 : Array to implement a mapping table

```
newtype CharstringToPid Array (Charstring, Pid)
endnewtype;
```

The above type can be used to map a Charstring representing a name to a Pid value representing the corresponding process instance.

## String

The `String` generator takes two generator parameters, the component sort and the name of an empty string value. A value of a `String` type is a sequence of component sort values. There is no restriction on the length of the sequence. The predefined sort `Charstring`, for example, is defined as an application of the `String` generator.

Example 19 : String generator

```
newtype S1 String(Integer, empty)
endnewtype;
```

Above, a `String` with `Integer` components is defined. An empty string, a string with the length zero, is represented by the literal `empty`.

The following operators are available in instantiations of `String`.

```
mkstring : Itemsort -> String
length : String -> Integer
first : String -> Itemsort
last : String -> Itemsort
"/" : String, String -> String
substring : String, Integer, Integer -> String
append : in/out String, Itemsort;
( . ) : * Itemsort -> String
```

In this enumeration of operators, `String` should be replaced by the string newtype (`s1` in the example above) and `Itemsort` should be replaced by the component sort parameter (`Integer` in the example above). The operators have the following behavior, with the examples based on type `String (Integer, empty)`:

- ✧ `mkstring` :  
This operator takes one `Itemsort` value and converts it to a `String` of length 1.  
`mkstring (-3)` gives a string of one integer with value -3.
- ✧ `length` :  
The number of elements, i.e. `Itemsort` values, in the `String` passed as parameter.  
`length (empty) = 0`, `length(mkstring (2)) = 1`
- ✧ `first` :  
The value of the first `Itemsort` element in the `String` passed as parameter. If the length of the `String` is 0, then it is an error to call the `first` operator.  
`first (mkstring (8) // mkstring (2)) = 8`
- ✧ `last` :  
The value of the last `Itemsort` element in the `String` passed as parameter. If the length of the `String` is 0, then it is an error to call the `last` operator.  
`last (mkstring (8) // mkstring (2)) = 2`
- ✧ `// (concatenation)` :  
The result is a `String` with all the elements in the first parameter, followed by all the elements in the second parameter.  
`mkstring (8) // mkstring(2)` gives a string of two elements: 8 followed by 2.
- ✧ `substring` :  
The result is a copy of a part of the `String` passed as first parameter. The copy starts at the index given as second parameter (Note: first `Itemsort` element has index 1). The length of

the copy is specified by the third parameter. It is an error to try to access elements outside of the true length of the first parameter.

```
substring (mkstring (8) // mkstring(2), 2, 1)
= mkstring(2)
```

≈ **append** :

**append** is a Telelogic extension and can be used to add a new component to the end of an existing String. **append** takes a variable as first parameter and a **Itemsort** value as second. The variable is then updated to include the second parameter as last component in the String. The reason for introducing this operator is that:

```
task append(V, Comp);
```

is much more efficient than performing the same calculation as

```
task V := V // mkstring(Comp);
```

≈ **(. .)**:

The **(. .)** expression, which is a Telelogic extension, is an application of the implicit make operator that is present in all strings. The make operator takes a sequence of **Itemsort** values and returns a String that contains these value in the order they are given.

`String_var := (. 1, 2, 3 .)` would give a string containing 1, 2 and 3.

It is also possible to access **Itemsort** elements in a String by indexing a String variable. Assume that **C** is a String instantiation variable. Then it is possible to write:

```
task C(2) := C(3);
```

This would mean that **Itemsort** element number 2 is assigned the value of **Itemsort** element number 3 in the variable **C**. NOTE that the first element in a String has index 1. It is an error to index a String outside of its length.

The String generator can be used to build lists of items of the same type, although some typical list operations are computationally quite expensive, like inserting a new element in the middle of the list.

## Powerset

The **Powerset** generator takes one generator parameter, the item sort, and implements a powerset over that sort. A **Powerset** value can be seen as: for each possible value of the item sort it indicates whether that value is member of the **Powerset** or not.

**Powersets** can often be used as an abstraction of other, more simple data types. A 32-bit word seen as a bit pattern can be modeled as a **Powerset** over a syntype of **Integer** with the range 0:31. If, for example, 7 is member of the powerset this means that bit number 7 is set.

Example 20 : Powerset generator

```
syntype SmallInteger = Integer
constants 0:31
endsyntype;

newtype P1 Powerset(SmallInteger)
endnewtype;
```

The only literal for a powerset sort is `empty`, which represents a powerset containing no elements. The following operators are available for a powerset sort (replace **Powerset** with the name of the newtype, **P1** in the example above, and **Itemsort** with the **Itemsort** parameter, **SmallInteger** in

the example):

```

"in" : Itemsort, Powerset -> Boolean
incl : Itemsort, Powerset -> Powerset
incl : Itemsort, in/out Powerset;
del  : Itemsort, Powerset -> Powerset
del  : Itemsort, in/out Powerset;
length : Powerset -> Integer
take  : Powerset -> Itemsort
take  : Powerset, Integer -> Itemsort
"<" : Powerset, Powerset -> Boolean
">" : Powerset, Powerset -> Boolean
"<=" : Powerset, Powerset -> Boolean
">=" : Powerset, Powerset -> Boolean
"and" : Powerset, Powerset -> Powerset
"or"  : Powerset, Powerset -> Powerset
( . ) : * Itemsort -> Powerset

```

These operators have the following interpretation (the examples are based on newtype  $P_1$  of the above example, and it is supposed that variable  $v0\_1\_2$  of  $P_1$  contains elements 0, 1, and 2):

- ⌘ in :

This operator tests if a certain value is member of the powerset or not.

3 in incl (3, empty) gives true;

3 in v0\_1\_2 gives false, 0 in v0\_1\_2 gives true.
- ⌘ incl :

Includes a value in the powerset. The result is a copy of the Powerset parameter with the Itemsort parameter included. To include a value that is already member of a powerset is a null-action.

incl (3, empty) gives a set with one element, 3,

incl (3, v0\_1\_2) gives a set with elements, 0, 1, 2, and 3.
- ⌘ incl (second operator) :

This operator is a Telelogic extensions added as it is more efficient than the standard incl. This operator updates a powerset variable with a new component value.

task incl(3, v0\_1\_2); means the same as

task v0\_1\_2 := incl(3, v0\_1\_2);
- ⌘ del :

Deletes a member in a powerset. The result is a copy of the Powerset parameter with the Itemsort parameter deleted. To delete a value that is not member of a powerset is a null-action.

del (0, v0\_1\_2) gives a set with element 1 and 2;

del (30, v0\_1\_2) = v0\_1\_2
- ⌘ del (second operator) :

This operator is a Telelogic extensions added as it is more efficient than the standard del operator. This operator updates a powerset variable by removing a component value.

task del(3, v0\_1\_2); means the same as

task v0\_1\_2 := del(3, v0\_1\_2);
- ⌘ length :

The number of elements in the powerset.

length (v0\_1\_2) = 3, length (empty) = 0
- ⌘ take (one parameter) :

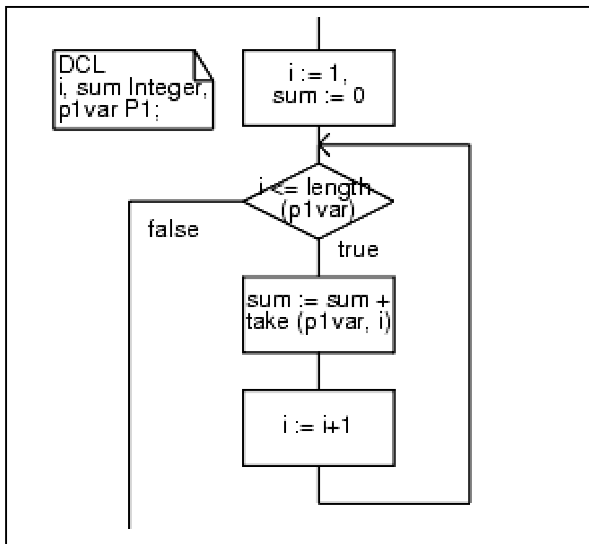
Returns one of the elements in the powerset, but it is not specified which one.

take (v0\_1\_2) gives 0, 1, or 2 (unspecified which of these three)

≪ take (two parameters) :

Elements are implicitly numbered with in the powerset from 1 to length(). The Telelogic-specific take operator returns the element with the number passed as second parameter. This operator can be used to "loop" through all elements of the set, as is illustrated in [Figure 27](#).

Figure 27 : Computing the sum of all elements in a Powerset



≪ < :

A<B, is A a true subset of B

incl (2, empty) < v0\_1\_2 = true,

incl (30, empty) < v0\_1\_2 = false

≪ > :

A>B, is B a true subset of A

≪ <= :

A<=B, is A a subset of B

≪ >= :

A>=B, is B a subset of A

≪ and :

Returns the intersection of the parameters, i.e. a powerset with the element members of both parameters.

incl (2, incl (4, empty)) and v0\_1\_2 gives a set with one element, visually 2.

≪ or :

Returns the union of the parameters, i.e. a powerset with the element members of any of the parameters.

incl (2, incl (4, empty)) or v0\_1\_2 gives a set with elements, 0, 1, 2, and 4.

≪ (. .) :

The (. .) expression, which is a Telelogic extension, is an application of the implicit make operator, that is present in all powersets. The make operator takes a sequence of Itemsort values and returns a Powerset that contains these values.

v0\_1\_2 := (. 1, 2, 3 .) would give a set including 1, 2 and 3.

Powerset resembles the Bag operator, and normally it is better to use Powerset. See also the discussion in ["Bag"](#).

## Bag

The Z.105-specific generator Bag is almost the same as Powerset. The only difference is that a bag

can contain the same value several times. In a Powerset a certain value is either member or not member of the set. A Bag instantiation contains the literal `empty` and the same operators, with the same behavior, as a Powerset instantiation. For details please see "[Powerset](#)".

A Bag contains one additional operator:

```
makebag : Itemsort -> Bag
```

≠ makebag :

Takes an Itemsort value and returns a Bag containing this value (length = 1).

It is recommended to use Powerset instead of Bag, except in cases where the number of instances of a value is important. Powerset is defined in Z.100, and is therefore more portable. Bag is mainly part of the predefined data types in order to support the ASN.1 `SET OF` construct.

### Ref, Own, Oref, Carray

These generators are Telelogic extensions to make it possible to work with pointers (Ref, Own, Oref) and with array with the same properties as in C.

Own and Oref is described in "[Own and ORef Generators](#)", while Ref and Carray is part of the package ctypes described in "[C Specific Package ctypes](#)". The package ctypes also contains SDL versions of some simple C types, which might be helpful in some cases.

### Literals

*Literals*, i.e. named values, can be included in newtypes.

Example 21 : Literals in struct newtype

```
newtype Coordinates struct
x integer;
y integer;
adding
literals Origo, One;
endnewtype;
```

In this struct there are two named values (literals); `Origo` and `One`. The only way in SDL to specify the values these literals represent is to use axioms. Axioms can be given in a section in a newtype. This is not further discussed here. The SDL to C compilers provide other ways to insert the values of the literals. Please see the documentation in [The Advanced/Cbasic SDL to C Compiler](#).

The literals can be used in SDL actions in the same way as expressions.

Example 22 : Use of literals

```
dcl C1 Coordinates;

task C1 := Origo;
decision C1 /= One;
...
```

Please note the differences in the interpretation of literals in the example above and in the description of enumeration types, see "[Enumeration Sorts](#)". In an enumeration type each literal introduces a new distinct value and the set of literals defines the possible values for the type. In the struct example above, the type and the set of possible values for the type is defined by the struct definition. The literals here only give names on already existing values.

An alternative that might be more clear, is to use literals in the case of an enumeration type and use operators without parameters (Telelogic extension) in other cases, like the struct above.

## Operators

*Operators* can be added to a newtype in the same way as literals.

Example 23 : Operators in struct newtype

```
newtype Coordinates struct
x integer;
y integer;
adding
operators
"+" : Coordinates, Coordinates -> Coordinates;
length : Coordinates -> Real;
endnewtype;
```

Telelogic has extended the operators with a number of new features to make them more flexible and to make it possible to have more efficient implementations. Extensions:

- ≈ in/out parameters
- ≈ operators without parameters
- ≈ operators without result

Example 24 : Operators

```
operators
op1 : in/out Coordinates;
op2 : -> Coordinates;
op3 : ;
```

In the example above `op1` takes one in/out parameter and has no result, `op2` has no parameters and returns a value of type `Coordinates`, while `op3` has neither parameters. nor result.

The behavior of operators can either be defined in axioms (as the literal values) or in operator diagrams. An operator diagram is almost identical to a value returning procedure (without states). An alternative to draw the operator implementation as a diagram is to define it in textual form. This might be appropriate as most operators performs calculations, and does not have anything to do with process control or process communication. In this case the algorithmic extension described in "[Compound Statement](#)" could be of great value.

Example 25 : Operator implementations

```
newtype Coordinates struct
```

```

x integer;
y integer;
adding
operators
"+" : Coordinates, Coordinates -> Coordinates;
operator "+" fpar a, b Coordinates
returns Coordinates
{
dcl result Coordinates;
result!x := a!x + b!x;
result!y := a!y + b!y;
return result;
}
endnewtype;

```

In the SDL to C Compilers there is also the possibility to include implementations in the target language. The problem with this is that it is necessary to know a lot more about the way the SDL to C Compilers translate operators into C.

## Default Value

In a newtype or syntype it is possible to insert a default clause stating the default value to be given to all variables of this type.

Example 26 : Default value in struct newtype

```

newtype Coordinates struct
x integer;
y integer;
default (. 0, 0 .);
endnewtype;

```

All variables of sort `Coordinates` will be given the initial value `(. 0, 0 .)`, except if an explicit default value is given for the variable in the variable declaration.

Example 27 : Explicit default value in variable declaration

```

dcl
C1 Coordinates := (. 1, 1 .),
C2 Coordinates;

```

Here `C1` has an explicit default value that is assigned at start-up. `C2` will have the default value specified in the newtype.

## Generators

It is possible in SDL to define *generators* with the same kind of properties as the pre-defined generators `Array`, `String`, `Powerset`, and `Bag`. As this is a difficult task and the support from the code generators is limited, it is not recommended for a non-specialist to try to define a generator.

The possibility to use user defined generators in the SDL to C Compilers is described in more

detail in ["Generators"](#).



<http://www.telelogic.com>

