9 HLA Literal Constants and Constant Expressions

9.1 HLA Literal Constants

Literal constants are those language elements that we normally think of as non-symbolic constant objects. HLA supports a wide variety of literal constants. The following sections describe those constants.

9.1.1 Numeric Constants

HLA lets you specify several different types of numeric constants.

9.1.1.1 Decimal Constants

The first and last characters of a decimal integer constant must be decimal digits (0..9). Interior positions may contain decimal digits and underscores. The purpose of the underscore is to provide a better presentation for large decimal values (i.e., use the underscore in place of a comma in large values). Example: 1_234_265.

Note: Technically, HLA does not allow negative literal integer constants. However, you can use the unary "-" operator to negate a value, so you'd never notice this omission (e.g., -123 is legal, it consists of the unary negation operator followed by a positive decimal literal constant). Therefore, HLA always returns type **unsXX** for all literal decimal constants. Also, note that HLA always uses a minimum size of **uns32** for literal decimal constants. If you absolutely, positively, want a literal constant to be treated as some other type, use one of the compile-time type coercion functions to change the type (e.g., **uns8**(1), **word**(2), or **int16**(3)). Generally, the type that HLA uses for the object is irrelevant since HLA will automatically promote a value to a larger or smaller type as appropriate.

Here are the following ranges for the various HLA unsigned data types:

uns8:	0..255
uns16:	0..65,535
uns32:	0..4,294,967,295
uns64:	0..18,446,744,073,709,551,615
uns128:	0..340,282,366,920,938,463,463,374,607,431,768,211,455

9.1.1.2 Hexadecimal Constants

Hexadecimal literal constants must begin with a dollar sign ("$") followed by a hexadecimal digit and must end with a hexadecimal digit (0..9, A..F, or a..f). Interior positions may contain hexadecimal digits or underscores. Hexadecimal constants are easiest to read if each group of four digits (starting from the least significant digit) is separated from the others by an underscore. E.g., $1A_2F34_5438.

If the constant fits into 32 bits or less, HLA always returns the **dword** type for a hexadecimal constant. For larger values, HLA will automatically use the **qword** or **lword** type, as appropriate. If you would like the hexadecimal value to have a different type, use one of the HLA compile-time type coercion functions to change the type (e.g., **byte**($12) or **qword**($54)).

Here are the following ranges for the various HLA hexadecimal data types:

uns8:	0..$FF
uns16:	0..$FFFF
uns32:	0..$FFFF_FFFF
uns64:	0..$FFFF_FFFF_FFFF_FFFF
uns128:	0..$FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF

9.1.1.3 Binary Constants

Binary literal constants begin with a percent sign ("%") followed by at least one binary digit (0/1) and they must end with a binary digit. Interior positions may contain binary digits or underscore

characters. Binary constants are easiest to read if each group of four digits (starting from the least significant digit) is separated from the others by an underscore. E.g., %10_1111_1010.

Like hexadecimal constants, HLA always associates the type **dword** with a "raw" binary constant; it will use the **qword** or **lword** type if the value is greater than 32 bits or 64 bits (respectively). If you want HLA to use a different type, use one of the compile-time type coercion functions to achieve this.

Obviously, binary constants may have as many binary digits as there are bits in the underlying type. This document will not attempt to write out the maximum binary literal constant!

9.1.1.4 Numeric Set Constants

HLA provides a special numeric constant form that lets you specify a numeric value by the bit positions containing ones. This corresponds to a *powerset* of integer values in the range 0..31. These constants take the following form:

```
@{ comma_separated_list_of_digits }
```

The *comma_separate_list_of_digits* can be empty (signifying no set bits, i.e., the value zero), a single digit, or a set of digits separated by commas. Here are some examples:

```
@{}
@{8}
@{1,2,8,24}
```

The corresponding numeric constant is given the type **dword** and is assigned the value that has ones in all the specified bit positions. For example, "@{8}" is equal to 256 since this value has a single set bit in bit position eight. Note that "@{0}" equals one, not zero (because the value one has a single set bit in position zero).

9.1.1.5 Real (Floating-Point) Constants

Floating-point (real) literal constants always begin with a decimal digit (never just a decimal point). A string of one or more decimal digits may be optionally followed by a decimal point and zero or more decimal digits (the fractional part). After the optional fractional part, a floating-point number may be followed by "e" or "E", a sign ("+" or "-"), and a string of one or more decimal digits (the exponent part). Underscores may appear between two adjacent digits in the floating-point number; their presence is intended to substitute for commas found in real-world decimal numbers.

Examples:

```
1.2
2.345e-2
0.5
1.2e4
2.3e+5
1_234_567.0
```

Literal real constants are always 80 bits and have the default type **real80**. If you wish to specify **real32** or **real64** literal constants, then use the **real32** or **real64** compile-time coercion functions to convert the values, e.g., **real32**(3.14159). During compile time, it's rare that you'd want to use one of the smaller types since they are less accurate at representing floating-point values (although this might be precisely why you decide to use the smaller real type, so the accuracy matches the computations you're doing at run-time).

The range of **real32** constants is approximately $10^{\pm 38}$ with 6-$^1/_2$ digits of precision; the range of **real64** values is approximately $10^{\pm 308}$ with approximately 14-$^1/_2$ digits of precision, and the range of **real80** constants is approximately $10^{\pm 4096}$ with about 18 digits of precision.

9.1.2 Boolean Constants

Boolean constants consist of the two predefined identifiers **true** and **false**. Note that your program may redefine these identifiers, but doing so is incredibly bad programming style. Since these are actual identifiers in the symbol table (and not reserved words), you must spell these identifiers in all lower case or HLA will complain (unlike reserved words that are case insensitive).

Internally, HLA represents **true** with one and **false** with zero. In fact, HLA's compile-time boolean operations only look at bit #0 of the boolean value (and always clear the other bits). HLA compile-time statements that expect a boolean expression do not use zero/not zero like C/C++ and a few other languages. Such expressions must have a boolean type with the values **true**/**false**; you cannot supply an integer expression and rely on zero/not zero evaluation as in C/C++ or BASIC.

9.1.3 Character Constants

Character literals generally consist of a single (graphic) character surrounded by apostrophes. To represent the apostrophe character use four apostrophes, e.g., ‘’’.

Another way to specify a character constant is by typing the "#" symbol followed by a numeric literal constant (decimal, hexadecimal, or binary). Examples: #13, #$D, #%1101.

9.1.4 Unicode Character Constants

Unicode character constants are 16-bit values. HLA provides limited support for Unicode literal constants. HLA supports the UTF/7 code point (character set), which is just the standard seven-bit ASCII character set and nine high-order zero bits. To specify a 16-bit literal Unicode constant simply prefix a standard ASCII literal constant with a 'u' or 'U', e.g.,

```
u'A'  - UTF/7 character constant for 'A'
```

Note that UTF/7 constants are simply the ASCII character codes zero extended to 16 bits.

HLA provides a second syntax for Unicode character constants that lets you enter values whose character codes are outside the range $20..$7E. You can specify a Unicode character constant by its numeric value using the 'u#nnnn' constant form. This form lets you specify a 16-bit value following the '#' in either binary, decimal, or hexadecimal form, e.g.,

```
u#1233

u#$60F

u%100100101001
```

9.1.5 String Constants

String literal constants consist of a sequence of (graphic) characters surrounded by quotes. To embed a quote within a string, insert a pair of quotes into the string, e.g., "He said ""This"" to me."

If two string literal constants are adjacent in a source file (with nothing but whitespace between them), then HLA will concatenate the two strings and present them to the parser as a single string. Furthermore, if a character constant is adjacent to a string, HLA will concatenate the character and string to form a single string object. This is useful, for example, when you need to embed control characters into a string, e.g.,

```
"This is the first line" #$d #$a "This is the second line" #$d #$a
```

HLA treats the above as a single string with a Windows newline sequence (CR/LF) at the end of each of the two lines of text.

9.1.6 Unicode String Constants

HLA lets you specify Unicode string literals by prefixing a standard string constant with a 'u' or a 'U'. Such string constants use the UTF/7 character set (that is, the standard ASCII character set) but reserve 16 bits for each character in the string. Note that the high order nine bits of each character in the string will contain zero.

As this was being written, there is no support for Unicode strings in the HLA Standard Library, though support for Unicode string functions should appear shortly (note that Windows' programmers can call the Unicode string functions that are part of the Windows' API).

9.1.7 Character Set Constants

A character set literal constant consists of several comma delimited character set expressions within a pair of braces. The character set expressions can either be individual character values or a pair of character values separated by an ellipse (".."). If an individual character expression appears

within the character set, then that character is a member of the set; if a pair of character expressions, separated by an ellipse, appears within a character set literal, then all characters between the first such expression and the second expression are members of the set. As a convenience, if a string constant appears between the braces, HLA will take the union of all the characters in that string and add those characters to the character set.

Examples:

```
{'a','b','c'}// a, b, and c.
{'a'..'c'}   // a, b, and c.
{'A'..'Z','a'..'z'}// Alphabetic characters.
{"cset"}         // The character set 'c', 'e', 's', and 't'.
{' ',#$d,#$a,#$9}// Whitespace (space, return, linefeed, tab).
```

HLA character sets are currently limited to holding characters from the 128-character ASCII character set. In the future, HLA may support an **`xcset`** type (supporting 256 elements) or even **wcset** (wide character sets), but that support does not currently exist.

9.2 Structured Constants

Structured constants are those whose data type is not a scalar. The structured constant types include array constants, record constants, union constants, and pointer constants.

9.2.1 Array Constants

HLA lets you specify an array literal constant by enclosing a set of values within a pair of square brackets. Since array elements must be homogenous, all elements in an array literal constant must be the same type or conformable to the same type. Examples:

```
[ 1, 2, 3, 4, 9, 17 ]
[ 'a', 'A', 'b', 'B' ]
[ "hello", "world" ]
```

Note that each item in the list of values can actually be a constant expression, not a simple literal value.

HLA array constants are always one-dimensional. This, however, is not a limitation because if you attempt to use array constants in a constant expression, the only thing that HLA checks is the total number of elements. Therefore, an array constant with eight integers can be assigned to any of the following arrays:

```
const
    a8:       int32[8]         := [1,2,3,4,5,6,7,8];
    a2x4:     int32[2,4]    := [1,2,3,4,5,6,7,8];
    a2x2x2:int32[2,2,2]:= [1,2,3,4,5,6,7,8];
```

Although HLA doesn't support the notation of a multi-dimensional array constant, HLA does allow you to include an array constant as one of the elements in an array constant. If an array constant appears as a list item within some other array constant, then HLA expands the interior constant in place, lengthening the list of items in the enclosing list. E.g., the following three array constants are equivalent:

```
[ [1,2,3,4], [5,6,7,8] ]
[ [ [1,2], [3,4] ], [ [5,6], [7,8] ] ]
[1,2,3,4,5,6,7,8]
```

Although the three array constants are identical, as far as HLA is concerned, you might want to use these three different forms to suggest the shape of the array in an actual declaration, e.g.,

```
const
    a8:        int32[8]          := [1,2,3,4,5,6,7,8];
    a2x4:      int32[2,4]     := [ [1,2,3,4], [5,6,7,8] ];
    a2x2x2:int32[2,2,2]:= [[[1,2], [3,4] ], [[5,6], [7,8]]];
```

Also note that symbol array constants, not just literal array constants, may appear in a literal array constant. For example, the following literal array constant creates a nine-element array holding the values one through nine at indexes zero through eight:

```
const Nine:int32[ 9 ]:= [ a8, 9 ];
```

This assumes, of course, that *a8* was previously declared as above. Since HLA "flattens" all array constants, you could have substituted *a2x4* or *ax2x2x* for *a8* in the example above and obtained identical results.

You may also create an array constant using the HLA **dup** operator. This operator uses the following syntax:

expression DUP [*expression_to_replicate*]

Where expression is an integer expression and *expression_to_replicate* is some expression, possibly an array constant. HLA generates an array constant by repeating the values in the *expression_to_replicate* the number of times specified by the expression. (Note: this does not create an array with expression elements unless the *expression_to_replicate* contains only a single value; it creates an array whose element count is expression times the number of items in the *expression_to_replicate*). Examples:

```
10 dup [1]   -- equivalent to [1,1,1,1,1,1,1,1,1,1]
5 dup [1,2]  -- equivalent to [1,2,1,2,1,2,1,2,1,2]
```

Please note that HLA does not allow class constants, so class objects may not appear in array constants. In addition, HLA does not allow generic pointer constants; only certain types of pointer constants are legal. See the discussion of pointer constants for more details.

9.2.2 Record Constants

HLA supports record constants using a syntax very similar to array constants. You enclose a comma-separated list of values for each field in a pair of square brackets. To differentiate array and record constants, the name of the record type and a colon must precede the opening square bracket, e.g.,

```
type
    Planet:
       record
          x  :int32;
          y  :int32;
          z  :int32;
          density:real64;
       endrecord;

const
    p  :Planet := Planet:[ 1, 12, 34, 1.96 ]
```

HLA associates the items in the list with the fields as they appear in the original record declaration. In this example, the values 1, 12, 34, and 1.96 are associated with fields *x, y, z,* and *density*, respectively. Of course, the types of the individual constants must match (or be conformable to) the types of the individual fields.

Note that you may not create a record constant for a particular record type if that record includes data types that cannot have compile-time constants associated with them. For example, if a field of a record is a class object, you cannot create a record constant for that type since you cannot create class constants.

9.2.3 Union Constants

Union constants allow you to initialize static union data structures in memory as well as initialize union fields of other data structures (including anonymous union fields in records). There are some important differences between HLA compile-time union constants and HLA run-time unions (as well as between the HLA run-time union constants and unions in other languages). Therefore, it's a good idea to begin the discussion of HLA's union constants with a description of these differences.

There are a couple of different reasons people use unions in a program. The original reason was to share a sequence of memory locations between various fields whose access is mutually exclusive. When using a union in this manner, one never reads the data from a field unless they've previous written data to that field and there are no intervening writes to other fields. The HLA compile-time language fully (and only) supports this use of union objects.

A second reason people use unions (especially in high-level languages) is to provide aliases to a given memory location; in particular, aliases whose data types are different. In this mode, a programmer might write a value to one field and then read that data using a different field (in order to access that data's bit representation as a different type). *HLA does not support this type of access to union constants.* The reason is quite simple: internally, HLA uses a special "variant" data type to represent all possible constant types. Whenever you create a union constant, HLA lets you provide a value for a single data field. From that point forward, HLA effectively treats the union constant as a scalar object whose type is the same as the field you've initialized; access to the other fields through the union constant is no longer possible. Therefore, you cannot use HLA compile-time constants to do type coercion; nor is there any need to since HLA provides a set of type coercion operators like **`@byte, @word, @dword, @int8,`** etc. As noted above, the main purpose for providing HLA union constants is to allow you to initialize static **union** variables; since you can only store one value into a memory location at a time, union constants only need to be able to represent a single field of the union at one time. Of course, at run time, you may access any field of the static union object you've created; but at compile-time you may only access the single field associated with a union constant.

An HLA literal union constant takes the following form:

```
typename.fieldname:[ constant_expression ]
```

The *typename* component above must be the name of a previously declared HLA union data type (i.e., a `union` type you've created in the `type` section). The *fieldname* component must be the name of a field within that union type. The *constant_expression* component must be a constant value (expression) whose type is the same as, or is automatically coercible to, the type of the *fieldname* field. Here is a complete example:

```
type
    u:union
        b:byte;
        w:word;
        d:dword;
        q:qword;
    endunion;

static
    uVar   :u     := u.w:[$1234];
```

The declaration for *uVar* initializes the first two bytes of this object in memory with the value $1234. Note that *uVar* is actually eight bytes long; HLA automatically zeros any unused bytes when initializing a static memory object with a union constant.

Note that you may place a literal union constant in records, arrays, and other composite data structures. The following is a simple example of a record constant that has a union as one of its fields:

```
type
    r   :record
```

```
            b:byte;
            uf:u;
            d:dword;
    endrecord;

static
    sr  :r := r:[0, u.d:[$1234_5678], 12345];
```

In this example, HLA initializes the *sr* variable with the byte value zero, followed by a double-word containing $1234_5678 and a double-word containing zero (to pad out the remainder of the union field), followed by a double-word containing 12345.

You can also create records that have anonymous unions in them and then initialize a record object with a literal constant. Consider the following type declaration with an anonymous union:

```
type
    rau :record
            b:byte;
            union
                c:char;
                d:dword;
            endunion;
            w:word;
    endrecord;
```

Because anonymous unions within a record do not have a type name associated with them, you cannot use the standard literal union constant syntax to initialize the anonymous union field (that syntax requires a type name). Instead, HLA offers you two choices when creating a literal record constant with an anonymous union field. The first alternative is to use the reserved word **union** in place of a *typename* when creating a literal union constant, e.g.,

```
static
    srau :rau := rau:[ 1, union.d:[$12345], $5678 ];
```

The second alternative is a shortcut notation. HLA allows you to simply specify a value that is compatible with the first field of the anonymous union and HLA will assign that value to the first field and ignore any other fields in the union, e.g.,

```
static
    srau2 :rau := rau:[ 1, 'c', $5678 ];
```

This is slightly dangerous since HLA relaxes type checking a bit here, but when creating tables of record constants, this is very convenient if you generally provide values for only a single field of the anonymous union; just make sure that the commonly used field appears first and you're in business.

Although HLA allows anonymous records within a union, there was no syntactically acceptable way to differentiate anonymous record fields from other fields in the union; therefore, HLA does not allow you to create union constants if the union type contains an anonymous record. The easy workaround is to create a named record field and specify the name of the record field when creating a union constant, e.g.,

```
type
    r   :record
            c:char;
            d:dword;
    endrecord;

    u   :union
```

```
            b:byte;
            x:r;
            w:word;
    endunion;

static
    y :u := u.x:[ r:[ 'a', 5]];
```

You may declare a union constant and then assign data to the specific fields as you would a record constant. The following example provides some samples of this:

```
type
    u_t :union
            b:byte;
            x:r;
            w:word;
    endunion;

val
    u :u_t;
        .
        .
        .
    ?u.b := 0;
        .
        .
        .
    ?u.w := $1234;
```

The two assignments above are roughly equivalent to the following:

```
    ?u := u_t.b:[0];

and

    ?u := u_t.w:[$1234];
```

However, to use the straight assignment (the former example) you must first declare the value *u* as a *u_t* union.

To access a value of a union constant, you use the familiar "dot notation" from records and other languages, e.g.,

```
    ?x := u.b;
        .
        .
        .
    ?y := u.w & $FF00;
```

Note, however, that you may only access the last field of the union into which you've stored some value. If you store data into one field and attempt to read the data from some other field of the union, HLA will report an error. Remember, you don't use union constants as a sneaky way to coerce one value's type to another (use the coercion functions for that purpose).

9.2.4 Pointer Constants

HLA allows a very limited form of a pointer constant. If you place an ampersand ("&") in front of a static object's name (i.e., the name of a **static** variable, **readonly** variable, **storage** variable, procedure, or iterator), HLA will compute the run-time offset of that variable. Pointer constants may not be used in arbitrary constant expressions. You may only use pointer constants in expressions used to initialize **static** or **readonly** variables or as constant expressions in 80x86 instructions. The following example demonstrates how pointer constants can be used:

```
program pointerConstDemo;

static
   t:int32;
   pt: pointer to int32 := &t;

begin pointerConstDemo;

   mov( &t, eax );

end pointerConstDemo;
```

```
Pointer constants also allow a fixed constant offset by appending "[
constant_expression ]" to the pointer constant, for example:
```

```
program pointerConstDemo;

static
   t:int32;
   pt: pointer to int32 := &t[2];

begin pointerConstDemo;

   mov( &t[-4], eax );

end pointerConstDemo;
```

These pointer constants are the address of the specified object plus an offset that is the number of bytes specified by the constant integer expression.

Also note that HLA allows the use of the reserved word **NULL** anywhere a pointer constant is legal. HLA substitutes the value zero for **NULL**. You can also use the HLA compile-time function **@pointer(n)** with an integer constant to tell HLA to treat that number as a pointer constant.

Note that you may obtain the address of the current location counter as a pointer constant by applying the "**&**" operator to the **@here** keyword, e.g.,

```
   mov( &@here, eax );
```

This places the address of the start of the instruction into EAX.

9.2.5 Regular Expression Constants

HLA uses compile-time "regex"-typed variables to hold compiled versions of regular expression. There is no literal form of a regular expression constant. The only way to generate a regular expression constant is in a **val, const**, or "?" declaration by assigning the "value" of a #regex macro declaration to a symbol, e.g.,

```
#regex someRegexMacro;
  <<regex macro body>>

#endregex
```

```
const
   compiledRegex :regex := someRegexMacro;
```

See the section on regular expressions in the chapter on The *HLA Compile-Time Language* for more details.

9.3 Constant Expressions in HLA

HLA provides a rich expression evaluator to process assembly-time expressions. HLA supports the following operators (sorting by decreasing precedence):

```
! (unary not),- (unary negation)
*, div, mod, /, <<, >>
+, -
=, = =, <>, !=, <=, >=, <, >
&, |, &, in
```

The following subsections describe each of these operators in detail.

9.3.1 Type Checking and Type Promotion

Many dyadic (two-operand) operators expect the types of their operands to be the same. Prior to actually performing such an operation, HLA evaluates the types of the operands and attempts to make them compatible. HLA uses a type algebra to determine if two (different) types are compatible; if they are not, HLA will report a type mismatch error during assembly. If the types are compatible, HLA will attempt to make them identical via *type promotion*. The type algebra describes how HLA promotes one type to another in order to make the two types compatible.

Usually, you can state a type algebra easily enough by providing "algebraic" type equations. For example, in high-level languages one could use a statement like "r = r + i" to suggest that the type of the resulting sum is real when the left operand is real and the right operand is integer (around the "+" operator). Unfortunately, HLA supports so many different data types and operators that any attempt to describe the type algebra in this fashion will produce so many equations that it would be difficult for the reader to absorb it all. Therefore, this document will rely on an informal English description of the type algebra to explain how HLA operates.

First, if two operands have the same basic type, but are different sizes, HLA promotes the smaller object to the same size as the larger object. Basic types include the following sets: {uns8, uns16, uns32, uns64, uns128}, {int8, int16, int32, int64, int128}, {byte, word, dword, qword, lword}, and {real32, real64, real80}[1]. So, if any two operands appear from one of these sets, then both operands are promoted to the larger of the two types.

If an unsigned and a signed operand appear around an operator, HLA produces a signed result. If the unsigned operand is smaller than the signed operand, HLA assigns both operands the signed type prior to the operation. If the unsigned and signed operands are the same size (or the unsigned operand is larger), HLA will first check the H.O. bit of the unsigned operand. If it is set, then HLA promotes the unsigned operand to the next larger signed type (e.g., **uns16** becomes **int32**). If the resulting signed type is larger than the other operand's type, it will be promoted as well. This scheme fails if you have an **uns128** value whose H.O. bit is set. In that case, HLA promotes both operands to **int128** and will produce incorrect results (because the **uns128** value just went negative when it's really positive). Therefore, you should attempt to limit unsigned values to 127 bits if you're going to be mixing signed and unsigned operations in the same expression.

Any mixture of hexadecimal types (**byte, word, dword, qword,** or **lword**) and an unsigned type produces an unsigned type; the size of the resulting unsigned type will be the larger of the two types. Likewise, any mixture of hexadecimal types and signed integer types will produce a signed integer whose size is the larger of the two types. This "strengthening" of the type (hexadecimal types are "weaker" than signed or unsigned types) may seem counter-intuitive to a die-hard

1. As this is being written, HLA doesn't fully support wchar or wstring types; ultimately the support will appear and you can add the sets {char, wchar} and {string, wstring} to the list.

assembly programmer; however, making the result type hexadecimal rather than signed/unsigned can create problems if the result has the H.O. bit set since information about whether the result is signed or unsigned would be lost at that point.

Mixing unsigned values and a **real32** value will produce a **real32** result or an error. HLA produces an error if the unsigned value requires more than 24 bits to represent exactly (which is the largest unsigned value you may represent within the **real32** format). Note that in addition to promoting the unsigned type to **real32**, HLA will also convert the unsigned value to a **real32** value. Promoting the type is not the same thing as converting the value; e.g., promoting **uns8** to **uns16** simply involves changing the type designation of the **uns8** object. HLA doesn't have to deal with the actual value because it keeps all values in an internal 128-bit format. However, the binary representation for unsigned and **real32** values is completely different, so HLA must do the value conversion as well. Note that if you really want to convert a value that requires more than 24 bits of precision to a **real32** object (with truncation), just convert the unsigned operand to **real64** or **real80** and then convert the larger operand to **real32** using the **real32**(expr) compile-time function. Since unsigned values are, well, unsigned and **real32** objects are signed, the conversion process always produces a non-negative value.

Mixing signed and **real32** values in an expression produces a **real32** result. Like unsigned operands, signed operands are limited to 24 bits of precision or HLA will report an error. Technically, you should get one more bit of precision from signed operands (since the **real32** format maintains its sign apart from the mantissa), but HLA still limits you to 24 bits during this conversion. If the signed integer value is negative, so will be the **real32** result.

If you mix hexadecimal and **real32** types, HLA treats the hexadecimal type as an unsigned value of the same size. See the discussion of unsigned and **real32** values earlier for the details.

If you mix an unsigned, signed, or hexadecimal type with a **real64** type, the result is an error (if HLA cannot exactly represent the value in **real64** format) or a **real64** result. The conversion is very similar to the **real32** conversion discussed above except you get 52 bits of integer precision rather than only 24 bits.

If you mix an unsigned, signed, or hexadecimal type with a **real80** type, the result is an error (if HLA cannot exactly represent the value in **real80** format) or a **real80** result. The conversion is very similar to the **real32** conversion discussed above except you get 64 bits of integer precision rather than only 24 bits. Note that conversion of integer objects 64-bits or less will always proceed without error; 128-bit values are the only ones that will get you into trouble.

If you mix a pair of different sized real values in the same expression, HLA will promote (and convert) the smaller real value to the same size as the larger real value.

The only non-numeric promotions that take place in an expression are between characters and strings. If a character and a string both appear in an expression, HLA will promote the character to a string of length one.

9.3.2 Type Coercion in HLA

HLA will report a type mismatch error if objects of incompatible types appear within an expression. Note that you may use the type-coercion compile-time functions to convert between types that HLA does not automatically support in an expression (see the discussion later in this document). You can also use the HLA type coercion operator to attach a specific type to a constant expression. The type coercion operator takes the following form:

```
(type typename constexpr)
```

The *typename* component must be a valid, declared type identifier (including any of the built-in types or appropriate user-defined types). The *constexpr* component can be any constant expression that is reasonably compatible with the specified type. "Reasonably compatible" means that the types are the same size or one of the primitive types. Examples:

```
(type int8 'a')
(type real32 constExpression+2)
(type boolean int8Val)
```

One important thing to remember is that type coercion is a bitwise operation. No conversion is done when coercing one type to another using this type coercion operation.

HLA also achieves type coercion using several compile-time functions. See the chapter on *The HLA Compile-Time Language* for more details on those type coercion functions.

9.3.3 !expr

The expression must be either boolean or a number. For boolean values, *!* computes the standard logical not operation. Numerically, HLA inverts only the L.O. bit of boolean values and clears the remaining bits of the boolean value. Therefore, the result is always zero or one when NOTting a boolean value (even if the boolean object errantly contained other set bits prior to the "!" operation). Remember, the "!" operator only looks at the L.O. bit; if the value was originally non-zero but the L.O. bit was clear[1], then "!" produces true. This is not a zero/not-zero operation.

For numbers, *!* computes the bitwise not operation on the bits of the number, that is, it inverts all the bits. The exact semantics of this operation depend upon the original data type of the value you're inverting. Therefore, the result of applying the "!" operator to an integer number may not always be intuitive because HLA always maintains 128-bits of precision, regardless of the underlying data type. Therefore, a full explanation of this operator's semantics must be given on a type-by-type basis.

uns8: Bits 8..127 of an Uns8 object are always zero. Therefore, when you apply the "!" operator to an Uns8 value, the result can no longer be an Uns8 object since bits 8..127 will now contain ones. Zeroing out the H.O. bits is not wise, because you could be assigning the result of this expression to a larger data type and you may very well expect those bits to be set. Therefore, HLA converts the type of "!u8expr" to type **byte** (which does allow the H.O. bits to contain non-zero values). If you assign an object of type **byte** to a larger object (e.g., type **word**), HLA will copy over the H.O. bits from the byte object to the larger object. Example:

```
val
    u8 :uns8 := 1;
    b8 := !u8; // produces $FFF..FFFE but registers as byte $FE.
    w16 :word := b8;// produces $FF..FFFE but registers as word $FFFE.
```

Note: If you really want to chop the value off at eight bits, you can use the compile-time byte function to achieve this, e.g.,

```
val
    u8 :uns8 := 1;
    b8 := byte(!u8);  // produces $FE.
    w16 :word := b8;  // produces $00FE.
```

uns16: The semantics are similar to **uns8** except, of course, applying "!" to an **uns16** value produces a **word** value rather than a **byte** value. Again, the "!" operator will set bits 16..127 to one in the result. If you want to ensure that the result contains no set bits beyond bit #15, use the compile-time **word** function to strip the value down to 16 bits (just like the **byte** function in the example above).

uns32: The semantics are similar to **uns8** except, of course, applying "!" to an **uns32** value produces a **dword** value rather than a **byte** value. Again, the "!" operator will set bits 32..127 to one in the result. If you want to ensure the result contains no set bits beyond bit #31 use the compile-time **dword** function to strip the value down to 32 bits (just like the **byte** function in the example above).

uns64: The semantics are similar to **uns8** except, of course, applying "!" to an **uns64** value produces a **qword** value rather than a **byte** value. Again, the "!" operator will set bits 64..127 to one in the result. If you want to ensure the result contains no set bits beyond bit #63 use the compile-time **qword** function to strip the value down to 64 bits (just like the **byte** function in the example above).

1. In theory, this should never happen since HLA maintains boolean values as zero or one.

uns128: Applying the "!" operator to an **uns128** object simply inverts all the bits. There are no funny semantics here. Resulting expression type is set to **lword**.

int8: Same semantics as **byte** (see explanation below).

int16: Same semantics as **word** (see explanation below).

int32: Same semantics as **dword** (see explanation below).

int64: Same semantics as **qword** (see explanation below).

int128: Applying the "!" operator to an **int128** object simply inverts all the bits. There are no funny semantics here. Resulting expression type is set to **lword**.

byte: Bits 8..127 of a byte (**int8**) value must all be zeros or all ones. The "!" operator enforces this. If any of the H.O. bits are non-zero, the "!" operator sets them all to zero in the result; if all of the H.O. bits are zero, the "!" operator sets the H.O. bits to ones in the result. Of course, this operator inverts bits 0..7 in the original value and returns this inverted result. Note that the type of the new value is always **byte** (even if the original sub-expression was **int8**).

word: Bits 16..127 of a **word** (**int16**) value must all be zeros or all ones. The "!" operator enforces this. If any of the H.O. bits are non-zero, the "!" operator sets them all to zero in the result; if all of the H.O. bits are zero, the "!" operator sets the H.O. bits to ones in the result. Of course, this operator inverts bits 0..15 in the original value and returns this inverted result. Note that the type of the new value is always **word** (even if the original sub-expression was **int16**).

dword: Bits 32..127 of a **dword** (**int32**) value must all be zeros or all ones. The "!" operator enforces this. If any of the H.O. bits are non-zero, the "!" operator sets them all to zero in the result; if all of the H.O. bits are zero, the "!" operator sets the H.O. bits to ones in the result. Of course, this operator inverts bits 0..31 in the original value and returns this inverted result. Note that the type of the new value is always **dword** (even if the original sub-expression was **int32**).

qword: Bits 64..127 of a **qword** (**int64**) value must all be zeros or all ones. The "!" operator enforces this. If any of the H.O. bits are non-zero, the "!" operator sets them all to zero in the result; if all of the H.O. bits are zero, the "!" operator sets the H.O. bits to ones in the result. Of course, this operator inverts bits 0..63 in the original value and returns this inverted result. Note that the type of the new value is always **qword** (even if the original sub-expression was **int64**).

lword: Applying the "!" operator to an **lword** object simply inverts all the bits. There are no funny semantics here..

No other types are legal with the "!" operator. HLA will report a type conflict error if you attempt to apply this operator to some other type.

If the operand is one of the integer types (signed, unsigned, hexadecimal), then HLA will set the type of the result to the smallest type within that class (signed, unsigned, or hexadecimal) that can hold the result (not including sign extension bits for negative numbers or zero extension bits for non-negative values).

9.3.4 - expr (unary negation operator)

The expression must either be a numeric value or a character set. For numeric values, "-" negates the value. For character sets, the "-" operator computes the complement of the character set (that is, it returns all the characters not found in the set).

Again, the exact semantics depend upon the type of the expression you're negating. The following paragraphs explain exactly what this operator does to its expression. For all integer values (**uns*XX*, int*XX*, byte, word, dword, qword,** and **lword**), the negation operator always does a full 128-bit negation of the supplied operand. The difference between these different data types is how HLA sets the resulting type of the expressions (as explained in the paragraphs below).

uns8: If the original value was in the range 128..255, then the resulting type is **int16**, otherwise the resulting type is **int8**. Because **uns8** values are always positive, the negated result is always negative, hence the result type is always a signed integer type.

uns16: If the original value was in the range 32678..65535, then the resulting type is **int32**, otherwise the resulting type is **int16**. Because **uns16** values are always positive, the negated result is always negative; hence, the result type is always a signed integer type.

uns32: If the original value was in the range $8000_0000..$FFFF_FFFF, then the resulting type is **int64**, otherwise the resulting type is **int32**. Because **uns32** values are always positive, the negated result is always negative; hence, the result type is always a signed integer type.

uns64: If the original value was in the range $8000_0000_0000_0000..$FFFF_FFFF_FFFF_FFFF, then the resulting type is **int128**, otherwise the resulting type is **int64**. Because **uns64** values are always positive, the negated result is always negative; hence, the result type is always a signed integer type.

uns128: The result type is always set to **int128**. Note that there is no check for overflow. Effectively, HLA treats **uns128** operands as though they were **int128** operands with respect to negation. So large positive (**uns128**) values become smaller unsigned values after the negation. If you need to mix and match 128-bit values in an expression, you should attempt to limit your unsigned values to 127 bits.

byte, int8,

word, int16,

dword, int32,

qword, int64,

lword,

int128: Negates the expression (full 128 bits) and assigns the original expression type to the result.

real32: Negates the **real32** value and returns a real32 result.

real64: Negates the **real64** value and returns a real64 result.

real80: Negates the **real80** value and returns a **real80** result.

cset: Computes the set complement (returns cset type). The set complement is all the items that were *not* previously in the set. Since HLA uses a bitmap representation for character sets, the complement of a character set is the same thing as inverting all the bits in the powerset.

If the operand is one of the integer types (signed, unsigned, hexadecimal), then HLA will set the type of the result to the smallest type within that class (signed, unsigned, or hexadecimal) that can hold the result (not including sign extension bits for negative numbers or zero extension bits for non-negative values).

9.3.5 expr1 * expr2

For numeric operands, the "*" operator produces their product. For character set operands, the "*"operator produces the intersection of the two sets. The exact result depends upon the types of the two operands to the "*" operator. To begin with, HLA attempts to make the types of the two operands identical if they are not already identical. HLA achieves this via type promotion (see the discussion earlier).

If the operands are unsigned or hexadecimal operands, HLA will compute their unsigned product. If the operands are signed, HLA computes their signed product. If the operands are real, HLA computes their real product. If the operands are integer (signed or unsigned) and less than (or equal to) 64 bits, HLA computes their exact result. If the operands are greater than 64 bits and their product would require more than 128 bits, HLA quietly overflows without error. Note that HLA always performs a 128-bit multiplication, regardless of the operands' sizes; however, objects that require 64 bits or less of precision will always produce a product that is 128 bits or less. HLA automatically extends the size of the result to the next greater size if the product will not fit into an integer that is the same size as the operands. HLA will actually choose the smallest possible size for the product (e.g., if the result only requires 16 bits of precision, the resulting type will be **uns16**, **int16**, or **word**). The resulting type is always unsigned if the operands were unsigned, signed if the operands were signed, and hexadecimal if the operands were hexadecimal.

If the operands are real operands, HLA computes their product and always produces a **real80** result. If you want to produce a smaller result via the '*' operator, use the **real32** or **real64** compile-time function to produce the smaller result, e.g., "real32(r32const * r32const2)". Note that all real arithmetic inside HLA is always performed using the FPU, hence the results are always **real80**. Other than trying to simulate the actual products a running program would produce, there is no real reason to coerce the product to a smaller value.

If the operands are character set operands, the '*' operator computes the intersection of the two sets. Since HLA uses a bitmap representation for character sets, this operator does a bitwise logical AND of the two 16-byte operands (this operation is roughly equivalent to applying the "&" operator to two **lword** operands).

If the operand is one of the integer types (signed, unsigned, hexadecimal), then HLA will set the type of the result to the smallest type within that class (signed, unsigned, or hexadecimal) that can hold the result (not including sign extension bits for negative numbers or zero extension bits for non-negative values).

9.3.6 expr1 div expr2

The two expressions must be integer (signed, unsigned, or hexadecimal) numbers. Supplying any other data type as an operand will produce an error. The **div** operator divides the first expression by the second and produces the truncated quotient result.

If the operands are unsigned, HLA will do a full 128/128-bit division and the resulting type will be unsigned (HLA sets the type to the smallest unsigned type that will completely hold the result). If the operands are signed, HLA will do a full 128/128 bit signed division and the resulting type will be the smallest **int*XX*** type that can hold the result. If the operands are hexadecimal values, HLA will do a full 128/128 bit unsigned division and set the resulting type to the smallest hex type that can hold the result.

Note that the **div** operator does not allow real operands. Use the "/" operator for real division.

HLA will set the type of the result to the smallest type within its class (signed, unsigned, or hexadecimal) that can hold the result (not including sign extension bits for negative numbers or zero extension bits for non-negative values).

9.3.7 expr1 mod expr2

The two expressions must be integer (signed, unsigned, or hexadecimal) numbers. The **mod** operator divides the first expression by the second and produces their remainder (this value is always positive).

If the operands are unsigned, HLA will do a full 128/128-bit division and return the remainder. The resulting type will be unsigned (HLA sets the type to the smallest unsigned type that will completely hold the result).

If the operands are signed, HLA will do a full 128/128 bit signed division and return the remainder. The resulting type will be the smallest **int*XX*** type that can hold the result.

If the operands are hexadecimal values, HLA will do a full 128/128 bit unsigned division and set the resulting type to the smallest hex type that can hold the result.

Note that the **mod** operator does not allow real operands. You'll have to define real modulus and write the expression yourself if you need the remainder from a real division.

HLA will set the type of the result to the smallest type within its class (signed, unsigned, or hexadecimal) that can hold the result (not including sign extension bits for negative numbers or zero extension bits for non-negative values).

9.3.8 expr1 / expr2

The two expressions must be numeric. The '/' operator divides the first expression by the second and produces their (**real80**) quotient result.

If the operands are integers (unsigned, signed, or hexadecimal) or the operands are **real32** or **real64**, HLA first converts them to **real80** before doing the division operation. The expression result is always **real80**.

9.3.9 expr1 << expr2

The two expressions must be integer (signed, unsigned, or hexadecimal) numbers. The second operand must be a small (32-bit or less) non-negative value in the range 0..128. The << operator shifts the first expression to the left the number of bits specified by the second expression. Note that

the result may require more bits to hold than the original type of the left operand. Any bits shifted out of bit position 127 are lost.

HLA will set the type of the result to the smallest type within the left operand's class (signed, unsigned, or hexadecimal) that can hold the result (not including sign extension bits for negative numbers or zero extension bits for non-negative values). Note that the '<<' operator can yield a smaller type (specifically, an eight bit type) if it shifts all the bits off the H.O. end of the number; generally, though, this operation produces larger result types than the left operand.

9.3.10 expr1 >> expr2

The two expressions must be integer (signed, unsigned, or hexadecimal) numbers. The second operand must be a small (32-bit or less) non-negative value in the range 0..128. The >> operator shifts the first expression to the right the number of bits specified by the second expression. Any bits shifted out of the L.O. bit are lost. Note that this shift is a *logical* shift right, not an *arithmetic* shift right (this is true even if the left operand is an **int*XX*** value). Therefore, this operation always shifts a zero into bit position 127.

Shift rights may produce a smaller type that the value of the left operand. HLA will always set the type of the result value to the minimum type size that has the same base class as the left operand.

9.3.11 expr1 + expr2

If the two expressions are numeric, the "+" operator produces their sum.

If the two expressions are strings or characters, the "+" operator produces a new string by concatenating the right expression to the end of the left expression.

If the two operands are character sets, the "+" operator produces their union.

If the operands are integer values (signed, unsigned, or hexadecimal), then HLA adds them together. Any overflow out of bit #127 (unsigned or hexadecimal) or bit #126 (signed) is quietly lost. HLA sets the type of the result to the smallest type size that will hold the sum; the type class (signed, unsigned, hexadecimal) will be the same as the operands. Note that it is possible for the type size to grow or shrink depending on the values of the operands (e.g., adding a positive and negative number could reduce the type size, adding two positive or two negative numbers may expand the result type's size).

When adding two real values (or a real and an integer value), HLA always produces a **real80** result.

Since HLA uses a bitmap to represent character sets, taking the union of two character sets is the same as doing a bitwise logical OR of all 16 bytes in the character set.

9.3.12 expr1 - expr2

If the two expressions are numeric, the "-" operator produces their difference.

If the two expressions are character sets, the "-" operator produces their set difference (that is, all the characters in *expr1* that are not also in *expr2*).

If the operands are integer values (signed, unsigned, or hexadecimal), then HLA subtracts the right operand from the left operand. Any overflow out of bit #127 (unsigned or hexadecimal) or bit #126 (signed) is quietly lost. HLA sets the type of the result to the smallest type size that will hold their difference; the type class (signed, unsigned, hexadecimal) will be the same as the operands. Note that it is possible for the type size to grow or shrink depending on the values of the operands (e.g., subtracting two negative or non-negative numbers could reduce the type size, subtracting a negative value from a non-negative value may expand the result type's size).

When subtracting two real values (or a real and an integer value), HLA always produces a **real80** result.

Since HLA uses a bitmap to represent character sets, taking the set of two character sets is the same as doing a bitwise logical AND of the left operand with the inverse of the right operand.

9.3.13 Comparisons (=, ==, <>, !=, <, <=, >, and >=)

```
expr1 = expr2
expr1 == expr2
expr1 <> expr2
expr1 != expr2
```

```
expr1 < expr2
expr1 <= expr2
expr1 > expr2
expr1 >= expr2
```

Note: "!=" and "<>" operators are identical. "=" and "==" operators are also identical.

The two expressions must be compatible (described earlier). These operators compare the two operands and return true or false depending upon the result of the comparison.

You may use the "=" and "<>" operators to compare two pointer constants (e.g., "&abc" or "&ptrVar[2]"). The other operators do not allow pointer constant operands.

All the above operators allow you to compare boolean values, enumerated values (types must match), integer (signed, unsigned, hexadecimal) values, character values, string values, real values, and character set values.

When comparing boolean values, note that **false < true**.

One character set is less than another is if it is a proper subset of the other. A character set is less than or equal to another set if it is a subset of that second set. Likewise, one character set is greater than, or greater than or equal to, another set if it is a proper superset, or a superset, respectively.

As with any programming language, you should take care when comparing two real values (especially for equality or inequality) as minor precision drifts can cause the comparison to fail.

9.3.14 expr1 & expr2

Note: "&&" and "&" mean different things to HLA. See the section on high-level language control structures for details on the "&&" operator.

The operands must both be boolean or they must both be numbers. With boolean operands the AND operator produces the logical and of the two operands (boolean result). With numeric operands, the AND operator produces the bitwise logical AND of the operands.

If the operand is one of the integer types (signed, unsigned, hexadecimal), then HLA will set the type of the result to the smallest type within that class (signed, unsigned, or hexadecimal) that can hold the result.

9.3.15 expr1 in expr2

The first expression must be a character value. The second expression must be a character set. The **in** operator returns **true** if the character is a member of the specified character set; it returns **false** otherwise.

9.3.16 expr1 | expr2

Note: "||" and "|" mean different things to HLA. See the section on high-level language control structures for details on the "||" operator.

The operands must both be boolean or they must both be numbers. With boolean operands the OR operator produces the logical OR of the two operands (boolean result). With numeric operands, the OR operator produces the bitwise or of the operands.

If the operand is one of the integer types (signed, unsigned, hexadecimal), then HLA will set the type of the result to the smallest type within that class (signed, unsigned, or hexadecimal) that can hold the result.

9.3.17 expr1 ^ expr2

The operands must both be boolean or they must both be numbers. With boolean operands the ^ operator produces the logical exclusive-or of the two operands (boolean result). With number operands, the ^ operator produces the bitwise exclusive-or of the operands.

If the operand is one of the integer types (signed, unsigned, hexadecimal), then HLA will set the type of the result to the smallest type within that class (signed, unsigned, or hexadecimal) that can hold the result.

9.3.18 (expr)

You may override the precedence of any operator(s) using parentheses in HLA constant expressions.

9.3.19 [comma_separated_list_of_expressions]

This produces an array expression. The type of the expression is an array type whose base element is the type of one of the expressions in the list. If there are two or more constant types in the array expression, HLA promotes the type of the array expression following the rules for mixed-mode arithmetic (see the rules earlier in this document).

9.3.20 record_type_name : [*comma separated list of field expressions*]

This produces a record expression. The expressions appearing within the brackets must match the respective fields of the specified record type. See the discussion earlier in this document.

9.3.21 identifier

An identifier is a legal component of a constant expression if the identifier's classification is **const** or **val** (that is, the identifier was declared in a constant or value section of the program). The expression evaluator substitutes the current declared value and type of the symbol within the expression. Constant expressions allow the following types:

```
boolean, enumerated types, uns8, uns16, uns32, uns64, uns128  byte,
word, dword, qword, lword, int8, int16, int32, int64, int128, char,
real32, real64, real80, string, and cset.
```

You may also specify arrays whose element base type is one of the above types (or a record or union subject to the following restriction). Likewise, you can specify record or union constants if all of their respective fields are one of the above primitive types or a value array, record, or union constant.

HLA allows array, record, and union constants. If you specify the name of an array, for example, HLA works with all the values of that array. Likewise, HLA can copy all the values of a record or union with a single statement.

HLA allows literal Unicode character and string constants (e.g., u'a' and u"unicode") or identifiers that are of **wchar** or **wstring** type in an expression, but no other terms are allowed in such an expression (as this is being written).

9.3.22 identifier1.identifier2 {...}

Selects a field from a **record** or **union** constant. *Identifier1* must be a record or union object defined in a **const** or **val** section. *Identifier2* (and any following dot-identifiers) must be a field of the record or union. HLA replaces this object with the value of the specified field.

Examples:

```
recval.fieldval
recval.subrecval.fieldval
```

Don't forget that with union constant, you may only access the last field into which you've actually stored data (see the section on union constants for more details).

9.3.23 identifier [index_list]

Identifier must be an array constant defined in either a **const** or **val** section. *Index_list* is a list of constant expressions separated by commas. The index list selects a specified element of the

"identifier" array. HLA reports an error if you supply more indices than the array has dimensions. HLA returns an array slice if you specify fewer indices than the array has dimensions (for example, if an array is declared as "a:uns8[4,4]" and you specify "a[2]" in a constant expression, HLA returns the third row of the array (a[2,0]..a[2,3]) as the value of this term).

Examples:

```
arrayval[0]
aval[1,4,0]
```