13 The HLA Compile-Time Language

13.1 HLA Compile-Time Language, Macros, and Pragmas

This topic section describes one of HLA's more impressive features - the compile time language. Combined with the macro preprocessor, the HLA compile-time language lets you customize the HLA language in almost an infinite variety of ways.

Compile-time programs are just that- programs that execute while HLA is compiling your source file. You embed compile-time language statements directly in your HLA source files and these short program fragments control how HLA compiles your assembly code.

This section doesn't fully explain the HLA compile-time language because you've already seen some major parts of it. For example, **val** constants in the HLA source file are equivalent to compile-time variables. The "?" statement is the compile-time assignment statement. This topic section, therefore, builds on the material that appears elsewhere in HLA Reference Manual.

13.2 Viewing the Output of the HLA Compile-Time Language

The HLA compile-time language can generate assembly language statements during the compilation of an HLA program. Because it isn't always obvious what code the compile-time language is generating, you'll sometimes need the ability to view the output of the HLA compiler. This is easily accomplished using the HLA command-line option "-hla". This command-line option tells HLA to produce an assembly language output file that uses a pseudo-HLA syntax. The result is not compilable under HLA, but it will show you the "pure" assembly language output that HLA produces from your original source file. Here's an example source file:

```
program demoCTLoutput;
var
   array:dword[11];
begin demoCTLoutput;

   #for( i := 0 to 10 )

      mov( i, array[ i*4 ] );

   #endfor;

end demoCTLoutput;
```

Here is the output that the HLA compiler produces for the main program when you supply the "-hla" command-line option:

```
begin _HLAMain;
procedure start;
begin start;
end start;

        call BuildExcepts__hla_;
        pushd( 0 );
        push( ebp );
        push( ebp );
        lea( [esp-4], ebp );
        sub( 44, esp );
        and( -4, esp );


        mov( 0, (type dword [ebp-48]) );
        mov( 1, (type dword [ebp-44]) );
        mov( 2, (type dword [ebp-40]) );
        mov( 3, (type dword [ebp-36]) );
        mov( 4, (type dword [ebp-32]) );
        mov( 5, (type dword [ebp-28]) );
        mov( 6, (type dword [ebp-24]) );
        mov( 7, (type dword [ebp-20]) );
        mov( 8, (type dword [ebp-16]) );
        mov( 9, (type dword [ebp-12]) );
        mov( 10, (type dword [ebp-8]) );
QuitMain__hla_::
        pushd( 0 );
        call( (type dword __imp__ExitProcess@4) );
end _HLAMain;
```

13.3 #linker Directive

The **#linker** directive passes a single string argument along to the linker. This is typically done to specify the name of some object or library file to link with the current file during the link edit phase. This directive has the following syntax:

```
#linker( "linker directive or file" )
```

For example, under Linux the following **#linker** commands tell HLA to have the linker link in part of the C Standard Library:

```
#if( os.linux )
   #linker( "-I /lib/ld-linux.so.2" )
   #linker( "-lc" )
#endif
```

As you can see from this example, each #linker string argument is really just a command-line argument that is passed along to the GNU ld linker (which is exactly how the **#linker** command operates under any OS). If multiple **#linker** commands appear in a source file (as is the case in this example), HLA concatenates the command-line arguments together (with a space separating them) prior to passing the command-line arguments to the linker.

13.4 The #Include Directive

Like most languages, HLA provides a source inclusion directive that inserts some other file into the middle of a source file during compilation. HLA's **#include** directive is very similar to the pragma of the same name in C/C++ and you primarily use them both for the same purpose: including library header files into your programs.

HLA's include directive has the following syntax:

```
#include( string_expression )
```

Note that any arbitrary compile-time string expression is legal. You are not limited to a literal string constant.

The **#include** directive is legal anywhere whitespace is legal. The string specifies a filename that HLA will insert into the program during compilation at the point the **#include** appears. If HLA cannot find the file specified by the string constant in the current directory (or in the directory specified if the string contains a pathname), then HLA tries to find the file in the location specified by the "hlainc" environment variable. If HLA still doesn't find the file, HLA will report an error.

Although you can use the **#include** directive to insert any arbitrary text at an arbitrary point in your program, the vast majority of the time you will use **#include** to include a library header file (either an HLA Standard Library header file or a library header file you've written) into your program. HLA requires that you compile all external files at lex level zero. Therefore, if you are including some declarations into your program, the **#include** directive should be just inside the main program. Convention dictates that **#include** directives that include library headers should appear immediately after the **program** or **unit** header in a file.

13.5 The #IncludeOnce Directive

When composing complex header files, particularly when constructing library header files, you may find in necessary to insert a **#include**("file") directive into some other header files. Generally, this is not a problem, HLA certainly allows nested include files (up to 256 files deep). However, unless you are very careful about how you organize your files, it is very easy to create an "include loop" where one header file includes another and that other header file includes the first. Attempting to compile a program that includes either header file results in an infinite "include loop" during compilation; clearly, this is not desirable.

The standard way to handle this situation is to surround all the statements in an include file with a **#if** statement as follows:

```
#if( !@defined( headerfilename_hhf ))

?headerfilename_hhf := true;

   << Statements associated with this header file go here >>

#endif
```

The first time HLA includes this file the symbol "headerfilename_hhf" is not defined, so HLA processes the statements in the body of the **#if** statement. The very first statement defines this "headerfilename_hhf" symbol (the value and type of this symbol are irrelevant for our purposes; only the fact that the symbol exists is important). Thereafter, if some other header file includes this file a second (or additional) time, the "headerfilename_hhf" symbol is defined, so HLA skips all the statements in the header file since the value of the boolean expression in the **#if** statement is false. Therefore, HLA only processes the statements of this header file (at least those inside the **#if** statement) the first time it encounters this particular header file.

A drawback to this scheme is that HLA must still open the header file and read every line from the file, even if it ignores all the lines in the file. For large header files, (e.g., the "stdlib.hhf" header file) this can consume a significant amount of time during compilation. The **#includeonce** directive provides a solution for this problem.

You use the **#includeonce** directive just like the **#include** directive. The only difference between the two is that HLA keeps track of all files it has processed using the **#include** or **#includeonce** directives and will not process a header file a second time if you attempt to include it using the **#includeonce** directive.

Whenever HLA processes the **#includeonce** directive, it first compares its string operand with a list of strings appearing in previous **#include** or **#includeonce** directives. If it matches one of these previous strings, then HLA ignores the **#includeonce** directive; if the include filename does not appear in HLA' internal list, then HLA adds this filename to the list and includes the file.

Note that HLA's **#includeonce** directive only compares strings for equality. If you use two separate filenames for the same file, HLA will not detect this and it will include the file a second time. E.g., if the current directory is "C:\hlafiles" then the following sequence will include the file "whoops.hhf" twice:

```
#IncludeOnce( "whoops.hhf" )
#IncludeOnce( "c:\whoops.hhf" )
```

Also note that the **#include** directive will include its file regardless of whether the program previously included that file with a **#includeonce** directive, e.g., the following sequence also includes "whoops.hhf" twice:

```
#IncludeOnce( "whoops.hhf" )
#Include( "whoops.hhf" )
```

For these two reasons, it's still a good idea to protect all header files using the **#if** technique mentioned earlier, even if you use the **#includeonce** directive throughout.

13.6 Macros

HLA has one of the most powerful macro expansion facilities of any programming language. HLA's macros are the key to extending the HLA language. The following subsections describe HLA's powerful macro processing facilities.

13.6.1 Standard Macros

HLA provides powerful macro capabilities. You can declare macros almost anywhere whitespace is allowed in a program using the following syntax:

```
#macro identifier ( optional_parameter_list ) ;
   statements
#endmacro
```

Note that a semicolon does not follow the `#endmacro` clause.

Example:

```
#macro MyMacro;
   ?i = i + 1;
#endmacro
```

The optional parameter list must be a list of one or more identifiers separated by commas. Unlike procedure declarations, you do not associate a type with macro parameters. HLA automatically associates the type "text" with macro parameters (except for two special cases noted below). Example:

```
#macro MacroWParms( a, b, c );
   ?a = b + c;
#endmacro
```

Optionally, the last (or only) name in the identifier list may take the form *`identifier[]`*. This syntax tells the macro that it may allow a variable number of parameters and HLA will create an array of string objects to hold all the parameters (HLA uses a string array rather than a text array because text arrays are illegal). Example:

```
#macro MacroWVarParms( a, b, c[] );
   ?a = b + @text( c[0]) + c[1] );
#endmacro
```

If the macro does not allow any parameters, then you follow the identifier with a semicolon (i.e., no parentheses or parameter identifiers). See the first example in this section for a macro without any parameters.

When using the array form (variable parameters) in a macro argument list, HLA will parse the remaining actual parameters and shove them into the array, one (perceived) parameter per string array element. Sometimes, however, you might want to handle the parameter parsing chores yourself (for example, to allow commas as part of an actual macro parameter) rather than have HLA handle this task for you. HLA provides an option to tell it to grab all remaining (or simply all) parameter text passed in the actual parameter list and stores all this data into a compile-time string object. To achieve this, you prefix the last (or only) formal macro parameter with the reserved word **string**, e.g.,

```
#macro MacroWStringParms( a, b, string c );
```

```
    <<macro body>>
#endmacro
```

In this example, the first two actual parameters will be assigned to the text objects *a* and *b* within the macro. Any remaining parameters will be collected as a single string and stored into the *c* formal parameter as a string.

One very useful purpose for string macro parameters is to allow you to grab a list of parameters you want to pass on to some other macro or procedure as a single object. E.g.,

```
procedure abc( a:byte; b:word; c:dword );
begin abc;
   .
   .
   .
end abc;

#macro CallsAbc( string abcParms );
   .
   .
   .
  abc( @text( abcParms ));
   .
   .
   .
#endmacro
   .
   .
   .
  CallsAbc( 1, 2, 3 );
```

The final macro invocation in this sequence passes the three parameters "1,2,3" to the *abc* function.

Occasionally you may need to define some symbols that are local to a particular macro invocation (that is, each invocation of the macro generates a unique symbol for a given identifier). The local identifier list allows you to do this. To declare a list of local identifiers, simply following the parameter list (after the parenthesis but before the semicolon) with a colon (":") and a comma separated list of identifiers, e.g.,

```
        #macro ThisMacro(parm1):id1,id2;
        ...
```

HLA automatically renames each symbol appearing in the local identifier list so that the new name is unique throughout the program. HLA creates unique symbols using some form such as `_XXXX_HLA_` where `XXXX` is some hexadecimal numeric value. To guarantee that HLA can generate unique symbols, you should avoid defining symbols of this form in your own programs (in general, symbols that begin and end with an underscore are reserved for use by the compiler and the HLA standard library). Example:

```
    #macro LocalSym : i,j;

    j:  cmp(ax, 0)
        jne( i )
        dec( ax )
        jmp( j )
    i:
    #endmacro
```

Without the local identifier list, multiple expansions of this macro within the same procedure would yield multiple statement definitions for `i` and `j`. With the local statement present, however, HLA substitutes symbols similar to `_0001_HLA_` and `_0002_HLA_` for `i` and `j` for the first invocation and symbols like `_0003_HLA_` and `_0004_HLA_` for `i` and `j` on the second invocation, etc. This avoids duplicate symbol errors if you do not use (poorly chosen) identifiers like `_0001_HLA_` and `_0004_HLA_` in your code.

The statements section of the macro may contain any legal HLA statements (including definitions of other macros). However, the legality of such statements is controlled by where you expand the macro.

To invoke a macro, you simply supply its name and an appropriate set of parameters. Unless you specify a variable number of parameters (using the array syntax) then the number of actual parameters must exactly match the number of formal parameters. If you specify a variable number of parameters, then the number of actual parameters must be greater than or equal to the number of formal parameters (not counting the array parameter).

During macro expansion, HLA automatically substitutes the text associated with an actual parameter for the formal parameter in the macro's body. The array parameter, however, is a string array rather than a text array so you will have force the expansion yourself using the **@text** function:

```
#macro example( variableParms[] );
   ?@text(variableParms[0]) := @text(variableParms[1]);
#endmacro
```

Actual macro parameters consist of a string of characters up to, but not including a separate comma or the closing parentheses, e.g.,

```
example( v1, x+2*y )
```

"`v1`" is the text for parameter #1, "`x+2*y`" is the text for parameter #2. Note that HLA strips all leading whitespace and control characters before and after the actual parameter when expanding the code in-line. The example immediately above would expand do the following:

```
?v1 := x+2*y;
```

If (balanced) parentheses appear in some macro's actual parameter list, HLA does not count the closing parenthesis as the end of the macro parameter. That is, the following is legal:

```
example(  v1, ((x+2)*y) )
```

This expands to:

```
?v1 := ((x+2)*y);
```

If you need to embed commas or unmatched parentheses in the text of an actual parameter, use the HLA literal quotes **#(** and **)#** to surround the text. Everything (except surrounding whitespace) inside the literal quotes will be included as part of the macro parameter's text. Example:

```
example( v1, #( array[0,1,i] )# )
```

The above expands to:

```
?v1 := array[0,1,i];
```

Without the literal quote operator, HLA would have expanded the code to

```
?V1 := array[0;
```

and then generated an error because (1) there were too many actual macro parameters (four instead of two) and (2) the expansion produces a syntax error.

Of course, HLA's macro parameter parser does not consider commas appearing inside string or character constants as parameter separators. The following is legal, as you would expect:

```
    example( charVar, ‘,’ )
```

As you may have noticed in these examples, a macro invocation does not require a terminating semicolon. Macro expansion occurs upon encountering the closing parenthesis of the macro invocation. HLA uses this syntax to allow a macro expansion *anywhere* in an HLA source file. Consider the following:

```
#macro funny( dest )
    , dest );
#endmacro

    mov( 0 funny( ax )
```

This code expands to "mov(0, ax);" and produces a legal machine instruction. Of course, this is a truly horrible example of macro use (the style is really bad), but it demonstrates the power of HLA macros in your program. This "expand anywhere" philosophy is the primary reason macro invocations do not end with a semicolon.

13.6.2 Where You Declare a Macro Affects its Visibility

You may declare a macro almost anywhere whitespace is allowed in a program. This increases the utility of macros as part of the HLA Compile-time Language. However, there are some issues of which you should be aware when declare macros at arbitrary points; this section will discuss those issues so you can avoid some pitfalls of this new flexibility.

First, unless you have good reason to do otherwise, you really should declare your macros in a declaration section of your program. Long-time HLA programmers have grown used to finding them there and, by placing your macros in a declaration section (e.g., wherever a procedure declaration is allowed), you'll make your programs easier to read because other programmers can look for such declarations in a few known locations. Arbitrarily scattering your macro declarations all over the place can make your programs harder to read. In addition, it should be understood that you must declare a macro before the first invocation.

Like other identifiers in an HLA program, macro identifiers have a scope that limits their visibility. If you declare a macro within a procedure, then that macro's identifier is only visible within that procedure and you cannot invoke (call) the macro outside of the procedure (that is, beyond the `end` statement associated with the procedure). Note that this is true even if you declare the macro in the body of the procedure, outside the procedure's declaration section, e.g.,

```
    procedure SomeProc;
    begin SomeProc;

        #macro mov0eax;
            mov( 0, eax )
        #endmacro

        mov0eax;  // legal here

    end SomeProc;

    mov0eax; // undefined symbol here.
```

If you declare a macro in a namespace or within an HLA class, you may invoke that macro from outside the namespace or class declaration by prefixing the macro identifier with the namespace or class or object identifier using the normal dot-notation for access to fields of the namespace or class. Note that you may invoke namespace or class macros within the namespace or class without the namespace prefix (just as you may access other symbol types within the namespace or class without the prefix).

You may also embed macro definitions within records and unions. However, when you do this HLA will insert the macro's symbol into the field list for the record or union. Because HLA does not provide a way to access anything other than variable fields of a record or union outside the declaration of that type, you will not be able to invoke the macro from outside the record or union declaration. However, you may invoke that macro within the same record/union declaration that contains the macro definition, e.g.,

```
type
   r :record

      i:int32;
      #macro inrec;
         k:int32;
      #endmacro
      j:int32;
      inrec;  // Legal expansion here
   endrecord;

var
   r.inrec; // this is not legal here.  Use a namespace or class to do
this.
```

Because of some limitations of the HLA implementation language (Flex/Bison), there is an important peculiarity you should know when declaring macros. In particular, HLA may process a macro declaration before it finishes processing whatever occurs immediately before the macro. Therefore, if the successful definition of a macro depends on whatever appears immediately before the macro, the declaration may fail. Though this is rare, it does occur occasionally. Should this happen to you, try an insert an innocuous syntactical item (like a semicolon) before the macro declaration.

13.6.3 Multi-part (Context Free) Macro Invocations:

HLA macros provide some very powerful facilities not found in other macro assemblers. One of the unique features that HLA macros provide is support for multi-part (or context-free) macro invocations. This feature is accessed via the `#terminator` and `#keyword` reserved words. Consider the following macro declaration:

```
program demoTerminator;

#include( "stdio.hhf" );

#macro InfLoop:TopOfLoop, LoopExit;
   TopOfLoop:
#terminator endInfLoop;
   jmp TopOfLoop;
   LoopExit:
#endmacro;

static
   i:int32;

begin demoTerminator;

   mov( 0, i );
   InfLoop
```

```
        stdout.put( "i=", i, nl );
        inc( i );

    endInfLoop;

end demoTerminator;
```

The #`terminator` keyword, if it appears within a macro, defines a second macro that is available for a one-time use after invoking the main macro. In the example above, the `endInfLoop` macro is available only after the invocation of the `InfLoop` macro. Once you invoke the `EndInfLoop` macro, it is no longer available (though the macro calls can be nested, more on that later). During the invocation of the #`terminator` macro, all local symbols declared in the main macro (`InfLoop` above) are available (note that these symbols are not available outside the macro body. In particular, you could refer to neither `TopOfLoop` nor `LoopExit` in the statements appearing between the `InfLoop` and `endInfLoop` invocations above). The code above, by the way, emits code similar to the following:

```
_0000_HLA_:
        stdout.put( "i=", i, nl );
        inc( i );
        jmp _0000_HLA_;
_0001_HLA_:
```

The macro expansion code appears in italics. This program, therefore, generates an infinite loop that prints successive integer values.

These macros are called multi-part macros for the obvious reason: they come in multiple pieces (note, though, that HLA only allows a single #`terminator` macro). They are also referred to as *Context-Free macros* because of their syntactical nature. Earlier, this document claimed that you could refer to the #`terminator` macro only once after invoking the main macro. Technically, this should have said "you can invoke the terminator once for each outstanding invocation of the main macro." In other words, you can nest these macro calls, e.g.,

```
    InfLoop

        mov( 0, j );
        InfLoop

            inc( i );
            inc( j );
            stdout.put( "i=", i, " j=", j, nl );

        endInfLoop;

    endInfLoop;
```

The term *Context-Free* comes from automata theory; it describes this nestable feature of these macros.

As should be painfully obvious from this `InfLoop` macro example, it would be nice if one could define more than one macro within this context-free group. Furthermore, the need often arises to define limited-scope scope macros that can be invoked more than once (limited-scope means between the main macro call and the terminator macro invocation). The #`keyword` definition allows you to create such macros.

In the `InfLoop` example above, it would be nice if you could exit the loop using a statement like `brkLoop` (note that **break** is an HLA reserved word and cannot be used for this purpose). The #`keyword` section of a macro allows you to do exactly this. Consider the following macro definition:

```
#macro InfLoop:TopOfLoop, LoopExit;
   TopOfLoop:
#keyword brkLoop;
   jmp LoopExit;
#terminator endInfLoop;
   jmp TopOfLoop;
   LoopExit:
#endmacro;
```

As with the #terminator section, the #keyword section defines a macro that is active after the main macro invocation until the terminator macro invocation. However, #keyword macro invocations do not terminate the multi-part invocation. Furthermore, #keyword invocations may occur more that once. Consider the following code that might appear in the main program:

```
   mov( 0, i );
   InfLoop

      mov( 0, j );
      InfLoop

         inc( j );
         stdout.put( "i=", i, " j=", j, nl );
         if( j >= 10 ) then

            brkLoop;

         endif

      endInfLoop;
      inc( i );
      if( i >= 10 ) then

         brkLoop;

      endif;

   endInfLoop;
```

The brkLoop invocation inside the "if(j >= 10)" statement will break out of the inner-most loop, as expected (another feature of the context-free behavior of HLA's macros). The brkLoop invocation associated with the "if(i >= 10)" statement breaks out of the outer-most loop. Of course, the HLA language provides the **forever..endfor** loop and the **break** and **breakif** statements, so there is no need for this InfLoop macro, nevertheless, this example is useful because it is easy to understand. If you are looking for a challenge, try creating a statement similar to the C/C++ switch/case statement; it is perfectly possible to implement such a statement with HLA's macro facilities, see the HLA Standard Library for an example of the switch statement implemented as a macro.

The discussion above introduced the #keyword and #terminator macro sections in an informal way. There are a few details omitted from that discussion. First, the full syntax for HLA macro declarations is actually:

```
#macro identifier ( optional_parameter_list ) :optional_local_symbols;
   statements

#keyword identifier ( optional_parameter_list ) :optional_local_symbols;
   statements
```

```
note: additional #keyword declarations may appear here

#terminator identifier ( optional_parameter_list )
:optional_local_symbols;
   statements
#endmacro
```

There are three things that should immediately stand out here: (1) you may define more than one `#keyword` within a macro. (2) `#keywords` and `#terminators` allow optional parameters. (3) `#keywords` and `#terminators` allow their own local symbols.

The scope of the parameters and local symbols isn't particularly intuitive (although it turns out that the scope rules are exactly what you would want). The parameters and local symbols declared in the main macro declaration are available to all statements in the macro (including the statements in the `#keyword` and `#terminator` sections). The `InfLoop` macro used this feature since the JMP instructions in the `brkLoop` and `endInfLoop` sections referred to the local symbols declared in the main macro. The `InfLoop` macro did not declare any parameters, but had they been present, the `brkLoop` and `endInfLoop` sections could have used those parameters as well.

Parameters and local symbols declared in a `#keyword` or `#terminator` section are local to that particular section. In particular, parameters and/or local symbols declared in a `#keyword` section are not visible in other `#keyword` sections or in the `#terminator` section.

One important issue is that local symbols in a multipart macro are visible in the main code between the start of the multipart macro and the terminating macro. That is, if you have some sequence like the following:

```
    InfLoop

       jmp LoopExit;

    endInfLoop;
```

The HLA substitutes the appropriate internal symbol (e.g., _xxxx_HLA_) for the `LoopExit` symbol. This is somewhat unintuitive and might be considered a flaw in HLA's design. Future versions of HLA may deal with this issue; in the meantime, however, some code takes advantage of this feature (to mask global symbols) so it's not easy to change without breaking a lot of code. Be forewarned before taking advantage of this "feature", however, that it will probably change in HLA v3.x. An important aspect of this behavior is that macro parameter names are also visible in the code section between the initial macro and the terminator macro. Therefore, you must take care to choose macro parameter names that will not conflict with other identifiers in your program. E.g., the following will probably lead to some problems:

```
static
   i:int32;

#macro parmi(i);
   mov( i, eax );
#terminator endParmi;
   mov( eax, i );
#endmacro
   .
   .
   .
   parmi( xyz );
   mov( i, ebx );// actually moves xyz into ebx, since the parameter i
                 // overrides the global variable i here.
   endParmi;
```

13.6.4 Macro Invocations and Macro Parameters:

As mentioned earlier, HLA treats all non-array macro parameters as text constants that are assigned a string corresponding to the actual parameter(s) passed to the macro. I.e., consider the following:

```
#macro SetI( v );
    ?i := v;
#endmacro

SetI( 2 );
```

The above macro and invocation is roughly equivalent to the following:

```
const
    v : text := "2";
    ?i := v;
```

When utilizing variable parameter lists in a macro, HLA treats the parameter object as a string array rather than a text array (because HLA does not support text arrays). For example, consider the following macro and invocation:

```
#macro SetI2( v[] );
    ?i := v[0];
#endmacro

SetI2( 2 );
```

Although this looks quite similar to the previous example, there is a subtle difference between the two. The former example will initialize the constant (value) `i` with the `int32` value 2. The second example will initialize `i` with the string value "2".

If you need to treat a macro array parameter as text rather than as a string object, use the HLA **@text** function that expands a string parameter as text. E.g., the former example could be rewritten as:

```
#macro SetI2( v[] );
    ?i := @text( v[0] );
#endmacro

SetI2( 2 );
```

In this example, the **@text** function tells HLA to expand the string value `v[0]` (which is "2") directly as text, so the "SetI2(2)" invocation expands as

```
?i := 2;
rather than as
?i := "2";
```

On occasion, you may need to do the converse of this operation. That is, you may want to treat a standard (non-array) macro parameter as a string object rather than as a text object. You can accomplish this by using the **@string(** ***text_object*** **)** function. When HLA encounters this construct, it will substitute a string constant for the identifier. The following example demonstrates one possible use of this feature:

```
program demoString;
```

```
#macro seti3( v );
   #print( "i is being set to " + @string( v ))
   ?i := v;
#endmacro


begin demoString;

   seti3( 4 )
   #print( "i = " + string( i ) )
   seti3( 2 )
   #print( "i = " + string( i ) )

end demoString;
```

Note that HLA supports a second, deprecated, form: **@string:***identifier.* Though you might see this form in older source code, you should not use this form in HLA v2.x programs as this feature will probably be eliminated from a future version of HLA.

If an identifier is a **text** constant (e.g., a macro parameter or a const/value identifier of type **text**), special care must be taken to modify the string associated with that text object. A simple **val** expression like the following won't work:

```
?textVar:text := "SomeNewText";
```

The reason this doesn't work is subtle: if `textVar` is already a text object, HLA immediately replaces `textVar` with its corresponding string; this includes the occurrence of the identifier immediately after the "?" in the example above. So were you to execute the following two statements:

```
?textVar:text := "x";
?textVar:text := "1";
```

the second statement would not change `textVar's` value from "x" to "1". Instead, the second statement above would be converted to:

```
?x:text := "1";
```

and `textVar's` value would remain "x". To overcome this problem, HLA provides a special syntactical entity that converts a text object to a string and then returns the text object ID. The syntax for this special form is **@tostring:***identifier*. The example above could be rewritten as:

```
?textVar:text := "x";
?@tostring:textVar:text := "1";
```

In this example, *textVar* would be a text object that expands to the string "1".

13.6.5 Processing Macro Parameters

As described earlier, HLA processes as parameters all text between a set of matching parentheses after the macro's name in a macro invocation. HLA macro parameters are delimited by the surrounding parentheses and commas. That is, the first parameter consists of all text beyond the left parenthesis up to the first comma (or up to the right parenthesis if there is only one parameter). The second parameter consists of all text just beyond the first comma up to the second comma (or right parenthesis if there are only two parameters); and so on. The last parameter consists of all text from the last comma to the closing right parenthesis. Within a single parameter, any text appearing between "(" and ")", "[" and "]", or "{" and "}" will be considered as a single parameter, even if there are commas present. Therefore, procedure calls (with parameters), array element accesses, character set constants, macro invocations, and other such items will constitute a single macro parameter. The follow macro invocation, for example, has three arguments because the *parmMacro* invocation counts as a single parameter:

```
    ThreeParms( a, parmMacro( 0, 1 ), b );
```

This example demonstrates another feature of HLA's macro processing system - HLA uses *deferred macro parameter expansion*. That is, the text of a macro parameter is expanded when HLA encounters the formal parameter within the macro's body, *not* while HLA is processing the actual parameters in the macro invocation (which would be *eager* evaluation).

There are three exceptions to the rule of deferred parameter evaluation: (1) text constants are always expanded in an eager fashion (that is, the value of the text constant, not the text constant's name, is passed as the macro parameter). (2) The **@text** function, if it appears in a parameter list, expands the string parameter in an eager fashion. (3) The **@eval** function immediately evaluates its parameter; the discussion of **@eval** appears a little later.

In general, there is very little difference between eager and deferred evaluation of macro parameters. In some rare cases, there is a semantic difference between the two. For example, consider the following two programs:

```
program demoDeferred;
#macro two( x, y ):z;
    ?z:text:="1";
    x+y
#endmacro

const
    z:string := "2";

begin demoDeferred;

    ?i := two( z, 2 );
    #print( "i=" + string( i ))

end demoDeferred;
```

In the example above, the code passes the actual parameter "`z`" as the value for the formal parameter "`x`". Therefore, whenever HLA expands "`x`" it gets the value "`z`", which is a local symbol inside the "two" macro, that expands to the value "`1`". Therefore, this code prints "`3`" ("`1`" plus `y's` value which is "`2`") during assembly. Now consider the following code:

```
program demoEager;
#macro two( x, y ):z;
    ?z:text:="1";
    x+y
#endmacro

const
    z:string := "2";

begin demoEager;

    ?i := two( @text( z ), 2 );
    #print( "i=" + string( i ))

end demoEager;
```

The only differences between these two programs are their names and the fact that `demoEager` invocation of "`two`" uses the **@text** function to eagerly expand `z's` text. As a result, the formal parameter "`x`" is given the value of `z's` expansion ("`2`") and HLA ignores the local value for "`z`" in macro "`two`". This code prints the value "`4`" during assembly. Note that changing "`z`" in the main program to a text constant (rather than a string constant) has the same effect:

```
program demoEager;
#macro two( x, y ):z;
   ?z:text:="1";
   x+y
#endmacro

const
   z:text := "2";

begin demoEager;

   ?i := two( z, 2 );
   #print( "i=" + string( i ))

end demoEager;
```

This program also prints "4" during assembly.

One place where deferred vs. eager evaluation can get you into trouble is with some of the HLA built-in functions. Consider the following HLA macro:

```
#macro DemoProblem( Parm );

   #print( string( Parm ) )

#endmacro
   .
   .
   .
DemoProblem( @linenumber );
```

(The **@linenumber** function returns, as an `uns32` constant, the current line number in the file).

When this program fragment compiles, HLA will use deferred evaluation and pass the text **@linenumber** as the parameter `Parm`. Upon compilation of this fragment, the macro will expand to "#print(string(@linenumber))" with the intent, apparently, being to print the line number of the statement containing the *DemoProblem* invocation. In reality, that is not what this code will do. Instead, it will print the line number, in the macro, of the "#print(string (Parm));" statement. By delaying the substitution of the current line number for the **@linenumber** function call until inside the macro, deferred execution produces the wrong result. What is really needed here is eager evaluation so that the **@linenumber** function expands to the line number string before being passed as a parameter to the `DemoProblem` macro. The **@eval** built-in function provides this capability. The following coding of the `DemoProblem` macro invocation will solve the problem:

```
DemoProblem( @eval( @linenumber ) );
```

Now the compiler will execute the **@linenumber** function and pass that number as the macro parameter text rather than the string "@linenumber". Therefore, the **#print** statement inside the macro will print the actual line number of the *DemoProblem* statement rather than the line number of the #print statement.

Keep these minor differences in mind if you run into trouble using macro parameters.

13.7 Built-in Functions:

HLA provides several built-in functions that take constant operands and produce constant results. It is important that you differentiate these compile-time functions from run-time functions. These functions do not emit any object code, and therefore do not exist while your program is running. They are only available while HLA is compiling your program. Note that many of these functions are trivial to implement in assembly language or have counterparts in the HLA standard library. Therefore, the fact that they are not available at run-time shouldn’t prove to be much of a problem.

13.8 Constant Type Conversion Functions

`boolean( const_expr )`

The expression must be an ordinal or string expression. If `const_expr` is numeric, this function returns **false** for zero and **true** for everything else. If `const_expr` is a character, this function returns true for "T" and false for "F". It generates an error for any other character value. If `const_expr` is a string, the string must contain "true" or "false" else HLA generates an error.

```
int8( const_expr )
int16( const_expr )
int32( const_expr )
int64( const_expr )
int128( const_expr )
uns8( const_expr )
uns16 const_expr )
uns32( const_expr )
uns64( const_expr )
uns128( const_expr )
byte( const_expr )
word( const_expr )
dword( const_expr )
qword( const_expr )
lword( const_expr )
```

These functions convert their parameter to the specified integer. For real operands, the result is truncated to form a numeric operand. For all other numeric operands, the result is ranged checked. For character operands, the ASCII code of the specified character is returned. For boolean objects, zero or one is returned. For string operands, the string must be a sequence of decimal characters that are converted to the specified type. Note that **byte**, **word**, and **dword, …,** types are synonymous with **uns8**, **uns16**, and **uns32, …,** for the purposes of range checking.

```
real32( const_expr )
real64( const_expr )
real80( const_expr )
```

These functions are similar to the earlier integer functions, except these functions produce the obvious real results. Only numeric and string parameters are legal.

`char( const_expr )`

`Const_expr` must be an ordinal or string value. This function returns a character whose ASCII code is that ordinal value. For strings, this function returns the first character of the string.

`string( const_expr )`

This function produces a reasonable string representation of the parameter. Almost all data types are legal.

`cset( const_expr )`

The parameter must be a character, string, or character set. For character parameters, this function returns the singleton set containing only the specified character. For strings, each character in the string is unioned into the set and the function returns the result. If the parameter is a character set, this function makes a copy of that character set.

13.8.1 Bitwise Type Transfer Functions

The type conversion functions of the previous section will automatically convert their operands from the source type to the destination type. Sometimes you might want to change the type of some object without changing its value. For many "conversions" this is exactly what takes place. For example, when converting an `uns8` object to an `uns16` value using the `uns16(---)` function, HLA does not modify the bit pattern at all. For other conversions, however, HLA may completely change the underlying bit pattern when doing the conversion. For example, when converting the `real32` value 1.0 to a `dword` value, HLA completely changes the underlying bit pattern ($3F80_0000) so that the `dword` value is equal to one. On occasion, however, you might actually want to copy the bits straight across so that the resulting `dword` value is $3F80_0000. The HLA bit-transfer type conversion compile-time functions provide this facility.

The HLA bit-transfer type conversion functions are the following:

```
@int8( const_expr )
@int16( const_expr )
@int32( const_expr )
@int64( const_expr )
@int128( const_expr )
@uns8( const_expr )
@uns16 const_expr )
@uns32( const_expr )
@uns64( const_expr )
@uns128( const_expr )
@byte( const_expr )
@word( const_expr )
@dword( const_expr )
@qword( const_expr )
@lword( const_expr )
@real32( const_expr )
@real64( const_expr )
@real80( const_expr )
@char( const_expr )
@cset( const_expr )
```

The above functions extract eight, 16, 32, 64, or 128 bits from the constant expression for use as the value of the function. Note that supplying a string expression as an argument isn't particularly useful since the functions above will simply return the address of the string data in memory while HLA is compiling the program. The `@byte` function provides an additional syntax with two parameters; see the next section for details.

```
@string( const_expr )
```

HLA string objects are pointers (in both the language as well as within the compiler). So simply copying the bits to the internal string object would create problems since the bit pattern probably is not a valid pointer to string data during the compilation. With just a few exceptions, what the **@string** function does is takes the bit data of its argument and translates this to a string (up to 16 characters long). Note that the actual string may be between zero and 16 characters long since the HLA compiler (internally) uses zero-terminated strings to represent string constants. Note that the first zero byte found in the argument will end the string.

If you supply a string expression as an argument to `@string`, HLA simply returns the value of the string argument as the value for the `@string` function. If you supply a text object as an argument to the `@string` function, HLA returns the text data as a string without first expanding the text value . If you supply a pointer constant as an argument to the `@string` function, HLA returns the string that HLA will substitute for the static object when it emits the assembly file.

13.8.2 General functions

```
@abs( numeric_expr )
```

Returns the absolute equivalent of the numeric value passed as a parameter.

`@byte( integer_expr, which )`

The `which` parameter is a value in the range 0..15. This function extracts the specified byte from the value of the `integer_expression` parameter. (This is an extension of the **@byte** type transfer function.)

`@byte( real32_expr, which )`

The `which` parameter is a value in the range 0..3. This function extracts the specified byte from the value of the `real32_expression` parameter.

`@byte( real64_expr, which )`

The `which` parameter is a value in the range 0..7. This function extracts the specified byte from the value of the `real64_expression` parameter.

`@byte( real80_expr, which )`

The `which` parameter is a value in the range 0..9. This function extracts the specified byte from the value of the `real80_expression` parameter.

`@ceil( real_expr )`

This function returns the smallest integer value larger than or equal to the expression passed as a parameter. Note that although the result will be an integer, this function returns a **real80** value.

`@cos( real_expr )`

The real parameter is an angle in radians. This function returns the cosine of that angle.

`@date`

This function returns a string of the form "YYYY/MM/DD" containing the current date.

`@env( string_expr )`

This function returns a string containing the value of a system environment variable (whose name you pass as the string parameter). If the specified environment variable does not exist, this function returns the empty string.

`@exp( real_expr )`

This function returns a **real80** value that is the result of the computation e**`real_expr` (i.e., e raised to the specified power).

`@extract( cset_expr )`

This function returns a character from the specified character set constant. Note that this function doesn't actually remove the character from the set, if you want to do that, then you will need to explicitly remove the character yourself. The following code demonstrates how to do this:

```
program extractDemo;

val
    c:cset := {'a'..'z'};

begin extractDemo;

    #while( c <> {} )

        ?b := @extract( c );
        #print( "b=" + b )
        ?c := c - {b};
    #endwhile

end extractDemo;
```

`@floor( real_expr )`

This function returns the largest integer value less than or equal to the supplied expression. Note that the returned result is of type **real80** even though it has no fractional component.

`@isalpha( char_expr )`

This function returns **true** if the character expression is an upper or lower case alphabetic character.

`@isalphanum( char_expr )`

This function returns **true** if the parameter is an alphabetic or numeric character. It returns false otherwise.

`@isdigit( char_expr )`

This function returns **true** if the character expression is a decimal digit.

`@islower( char_expr )`

This function returns **true** if the character expression is a lower case alphabetic character.

`@isspace( char_expr )`

This function returns **true** if the character expression is a "whitespace" character. Typically, this would be spaces, tabs, newlines, returns, linefeeds, etc.

`@isupper( char_expr )`

This function returns **true** if the character expression is an upper case alphabetic character.

`@isxdigit( char_expr )`

This function returns **true** if the supplied character expression is a hexadecimal digit.

`@log( real_expr )`

This function returns the natural (base e) logarithm of the supplied parameter.

`@log10( real_expr )`

This function returns the base-10 logarithm of the supplied parameter.

`@max( comma_separated_list_of_ordinal_or_real_values )`

This function returns the largest value from the specified list.

`@min( comma_separated_list_of_ordinal_or_real_values )`

This function returns the least of the values in the specified list.

`@odd( int_expr )`

This function returns **true** if the integer expression is an odd number.

`@random( int_expr )`

This function returns a random **uns32** value.

`@randomize( int_expr )`

This function uses the integer expression passed as a parameter as the new seed value for the random number generator.

`@sin( real_expr )`

This function returns the sine of the angle (in radians) passed as a parameter.

`@sort(array_expr, int_expr, left_compare_id, right_compare_id, str_expr)`

This function returns an array containing the elements of *array_expr* sorted in ascending order. The second parameter (*int_expr*) specifies the number of elements in the array to sort (sorting always begins with element zero and continues for *int_expr* elements). Note that **@sort** always returns an array that is the same size as *array_expr*, but only the first *int_expr* elements are sorted.

Because *array_expr* elements can be an arbitrary type, you must supply a mechanism for comparing individual elements of the array. This is accomplished using the last three parameters to **@sort.** First, you must supply the names of two HLA **val** objects as the `left_compare_id` and `right_compare_id` parameters. These two value objects must be the same type as an element of *array_expr*. The last parameter must be a string constant holding the name of a macro that will compare the values in these two identifiers and return true if *left_compare_id* is less than *right_compare_id* (This has to be a string constant so that HLA won't attempt to immediately expand the macro when encountering the name).

Though it shouldn't matter much, the current implementation of **@sort** uses a quick-sort algorithm. There is no guarantee that this function will continue to use quicksort in the future, however.

Here's a quick example:

```
#macro abcmp;
    (a < b)
#endmacro

val
    a:int32;
    b:int32;
    array:int32[8] := [8,7,6,5,4,3,2,1];
    sortedArray:int32[8] := @sort( array, @elements(array), a, b, "abcmp"
);
```

`@sqrt( real_expr )`

This function returns the square root of the parameter.

`@system( string_expr )`

This function executes the system command specified by the string (i.e., a command-line operation for a shell interpreter). It captures all the output sent to the standard output device by this command and returns that data as a string value.

`@tan( real_expr )`

This function returns the tangent of the angle (in radians) passed as a parameter.

`@time`

This function returns a string of the form "HH:MM:SS xM" (x= A or P) denoting the time at the point this function was called (according to the system clock).

13.8.3 String functions:

`@delete( str_expr, int_start, int_len )`

This function returns a string consisting of the `str_expr` passed as a parameter with (possibly) some characters removed. This function removes `int_len` characters from the string starting at index `int_start` (note that strings have a starting index of zero).

`@index( str_expr1, int_start, str_expr2 )`

This function searches for `str_expr2` within `str_expr1` starting at character position `int_start` within `str_expr1`. If the string is found, this function returns the index into `str1_expr1` of the first match (starting at `int_start`). This function returns -1 if there is no match.

`@insert( str_expr1, int_start, str_expr2 )`

This function inserts `str_expr2` into `str_expr1` just before the character at index `int_start`.

`@left( str_expr, int_length )`

This function returns the left-most `int_length` characters of the specified string.

`@leftdel( str_expr, int_length )`

This function deletes the left-most `int_length` characters of the specified string and returns the result.

`@length( str_expr )`

This function returns the length of the specified string.

`@lowercase( str_expr, int_start )`

This function returns a string of characters from `str_expr` with all uppercase alphabetic characters converted to lower case. Only those characters from `int_start` on are copied into the result string.

`@replace( str_expr1, str_expr2, str_expr3)`

This function searches for every occurrence of *str_expr2* found within *str_expr1*. It replaces each occurrence found with *str_expr3* and returns the resultant string.

`@replace( str_expr1, str_array_expr2 )`

An extended version of the `@replace` function. The second argument must be an arra of strings with an even number of elements. For each pair of strings this function will replace each occurrence of the first string of the pair found in `str_expr1` with the value of the second string in the pair. E.g., ?resultStr := @replace(“Hello there world”, [[“there”, “”], [“ world”, “world”]]);

`@right( str_expr, int_length )`

This function returns the right-most `int_length` characters of the specified string.

`@rightdel( str_expr, int_length )`

This function deletes the right-most `int_length` characters of the specified string and returns the result.

`@rindex( str_expr1, int_start, str_expr2 )`

Similar to the index function, but this function searches for the last occurrence of `str_expr2` in `str_expr1` rather than the first occurrence.

`@strbrk( str_expr, int_start, cset_expr )`

This function returns the index of the first character beyond `int_start` in `str_expr` that is a member of the `cset_expr` parameter. This function returns -1 if none of the characters are in the set.

`@strset(` *`char_expr, int_len`* `)`

This function returns a string consisting of `int_len` copies of `char_expr`.

`@strspan(` *`str_expr, int_start, cset_expr`* `)`

This function returns the index of the first character beyond position `int_start` in `str_expr` that is not a member of the `cset_expr` parameter. This function returns -1 if all of the characters are in the set.

`@substr(` *`str_expr, int_start, int_len`* `)`

This function returns the substring specified by the starting position and length in `str_expr`.

`@tokenize(` *`str_expr, int_start, cset_delims, cset_quotes`* `)`

This function returns an array of strings obtained by doing a lexical scan of the `str_expr` passed as a parameter (starting at character index `int_start`). Each element of this array consists of all characters between any sequences of delimiter characters (specified by the `cset_delims` parameter). The only exceptions are strings appearing between bracketing (quoting) symbols. The fourth parameter specifies the possible bracketing characters. If `cset_quotes` contains a quotation mark (") then all sequences of characters between a pair of quotes will be treated as a single string. Similarly, if `cset_quotes` contains an apostrophe, then all characters between a pair of apostrophes will be treated as a single string. If the `cset_quotes` parameters contains one of the pairs "(" / ")", "{" / "}", or "[" / "]" (both characters from a given pair must be present), then `@Tokenize` will consider all characters between these bracketing symbols to be a single string.

You should use the *@elements* function to determine how many strings are present in the resulting array of strings (this will always be a one-dimensional array, although it is possible for it to have zero elements).

`@trim( str_expr, int_start )`

This function returns a string consisting of the characters in `str_expr` starting at position `int_start` with all leading and trailing whitespace removed.

`@uppercase(` *`str_expr, int_start`* `)`

This function returns a string consisting of the characters in `str_expr` starting at position `int_start` with all lower case alphabetic character converted to uppercase.

13.8.4 String/Pattern matching functions

The HLA string/pattern matching functions all attempt to match a string against a pattern. These functions all return a boolean result indicating success or failure (i.e., whether the string matches the pattern).

Most of these functions have two optional parameters: `Remainder` and `Matched`. If the function succeeds it generally copies the matched string into the **val** string constant specified by the `Matched` parameter and it copies all the characters in the `InputStr` parameter the follow the matched text into the `Remainder` parameter. You may specify the `Remainder` parameter without also specifying the `Matched` parameter, but if you need the *matched* result, you must specify all the parameters. The `Remainder` and `Matched` parameters appear in italics in all of the following functions to denote that they are optional.

If the function fails, the values of the `Remainder` and `Matched` parameters are generally undefined.

`@peekCset( InputStr, charSet,` *`Remainder, Matched`* `)`

This function checks the first character of `InputStr` to see if it is a member of `charSet`. The function returns **true/false** depending on the result of the set membership test. If the function succeeds it copies the value of the `InputStr` parameter to `Remainder` and creates a single character string from the first character of `InputStr` and stores this into `Matched`.

`@oneCset( InputStr, charSet, *Remainder*, *Matched* )`

This function checks the first character of `InputStr` to see if it is a member of `charSet`. The function returns **true/false** depending on the result of the set membership test. If the function succeeds, it copies all characters but the first of `InputStr` parameter to `Remainder` and copies the first character of `InputStr` into `Matched`.

`@uptoCset( InputStr, charSet, *Remainder*, *Matched* )`

This function matches all characters up to, but not including, a single character from the `charSet` character set parameter. If the `InputStr` parameter does not contain a character in the specified character set, this function fails. If it succeeds, it copies all the matched characters (not including the character in the character set) to the `Matched` parameter and it copies all remaining characters (including the character in the character set) to the `Remainder` parameter.

`@zeroOrOneCset( InputStr, charSet, *Remainder*, *Matched* )`

If the first character of `InputStr` is a member of `charSet`, this function succeeds and returns that character in the `Matched` parameter. It also returns the remaining characters in the string in the `Remainder` parameter.

This function always succeeds (since it matches zero characters). If the first character of `InputStr` is not in `charSet`, then this function returns `InputStr` in *Remainder* and returns the empty string in `Matched`.

`@exactlynCset( InputStr, charSet, n, *Remainder*, *Matched* )`

This function returns true if the first `n` characters of `InputStr` are in the character set specified by `charSet`. The `n+1st` character must not be in the character set specified by `charSet`. If this function succeeds (i.e., returns true), then it copies the first `n` characters to the `Matched` string and it copies all remaining characters into the `Remainder` string. If this function fails and returns **false**, `Remainder` and `Matched` are undefined.

`@firstnCset( InputStr, charSet, n, *Remainder*, *Matched* )`

This function is very similar to `exactlyncset` except it doesn't require that the `n+1st` character not be a member of the `charSet` set. If the first `n` characters of `InputStr` are in `charSet`, this function succeeds (returning **true**) and copies those `n` characters into the `Matched` string; it also copies any following characters into the `Remainder` string.

`@nOrLessCset( InputStr, charSet, n, *Remainder*, *Matched* )`

This function always succeeds. It will match between zero and `n` characters in `InputStr` from the `charSet` set. The `n+1st` character may be in `charSet`, this function doesn't care and only matches up to the `nth` character. This function copies up to `n` matched characters to the `Matched` string (the empty string if it matches zero characters); the remaining characters in the string are copied to the `Remainder` parameter.

`@nOrMoreCset( InputStr, charSet, n, *Remainder*, *Matched* )`

This function succeeds if it matches at least `n` characters from `InputStr` against the `charSet` set. It returns **false** if there are fewer than `n` characters from `charSet` at the beginning of `InputStr`. If this function succeeds, it copies the characters it matches to the `Matched` string and all characters after that sequence to the `Remainder` string.

`@ntomCset( InputStr, charSet, n, *Remainder*, *Matched* )`

This function succeeds if `InputStr` begins with at least `n` characters from `charSet`. If additional characters in `InputStr` are in this set, `ntomcset` will match up to `m` characters (`n < m`). It will not match any additional characters beyond the `mth` character, although those characters may be in the `charSet` set without affecting the success/failure of this routine. If this routine succeeds, it copies all the characters it matches to the `Matched` parameter and any remaining characters to the `Remainder` parameter.

`@exactlyntomCset( InputStr, charSet, n,` *`Remainder`*`,` *`Matched`* `)`

Similar to the `ntomcset` function, except this function fails if more than `m` characters at the beginning of `InputStr` are in the specified character set.

`@zeroOrMoreCset( InputStr, charSet,` *`Remainder`*`,` *`Matched`* `)`

This function always succeeds. If the first character of `InputStr` is not in `charSet`, this function copies `InputStr` to `Remainder`, sets matched to the empty string, and returns **true**. If some sequence of characters at the beginning of *InputStr* is in `charSet`, this function copies those characters to `Matched` and copies the following characters to `Remainder`.

`@oneOrMoreCset( InputStr, charSet,` *`Remainder`*`,` *`Matched`* `)`

This function succeeds if `InputStr` begins with at least one character from `charSet`. It will match all characters at the beginning of `InputStr` that are members of `charSet`. It copies the matched chars to the `Matched` string and any remaining characters to the `Remainder` string. It fails if the first character of `InputStr` is not a member of *charSet.*

`@peekChar( InputStr, Character,` *`Remainder`*`,` *`Matched`* `)`

This function succeeds if the first character of `InputStr` matches `Character`. If it succeeds, it copies the character to the `Matched` string and copies the entire string (including the first character) to `Remainder`.

`@oneChar( InputStr, Character,` *`Remainder`*`,` *`Matched`* `)`

This function succeeds if the first character if `InputStr` is equal to `Character`. If it succeeds, it copies the matched character to `Matched` and any remaining characters to `Remainder`. If it fails, then `Remainder` and `Matched` are undefined.

`@uptoChar( InputStr, Character,` *`Remainder`*`,` *`Matched`* `)`

This function matches all characters up to, but not including, the specified character. If fails if the specified character is not in the `InputStr` string. If this function succeeds and returns **true**, it copies the matched character to the `Matched` string and copies all remaining characters to the `Remainder` string (the `Remainder` string will begin with the value found in `Character`). If this function fails, it leaves `Remainder` and `Matched` undefined.

`@zeroOrOneChar( InputStr, Character,` *`Remainder`*`,` *`Matched`* `)`

This function always succeeds since it can match zero characters. If the first character of `InputStr` is not equal to `Character` this function returns **true** and sets `Remainder` equal to `InputStr` and sets `Matched` to the empty string. If the first character of `InputStr` is equal to `Character`, then this function returns that character in `Matched` and returns any remaining characters from `InputStr` in `Remainder`.

`@zeroOrMoreChar( InputStr, Character,` *`Remainder`*`,` *`Matched`* `)`

This function always succeeds since it can match zero characters. If the first character of `InputStr` is not equal to `Character`, this function returns **true** and sets `Remainder` equal to `InputStr` and `setsMatched` to the empty string. If `InputStr` begins with a sequence of characters that are all equal to `Character`, then this function returns those characters in `Matched` and returns any remaining characters from `InputStr` in `Remainder`.

`@oneOrMoreChar( InputStr, Character, *Remainder*, *Matched* )`

This function always succeeds since it can match zero characters. If the first character of `InputStr` is not equal to `Character` this function returns **true** and sets `Remainder` equal to `InputStr` and sets `Matched` to the empty string. If `InputStr` begins with a sequence of characters that are all equal to `Character`, then this function returns those characters in `Matched` and returns any remaining characters from `InputStr` in `Remainder`.

`@exactlynChar( InputStr, Character, n, *Remainder*, *Matched* )`

This function returns true if the first `n` characters of `InputStr` are equal to `Character`. The `n+1st` character cannot be equal to `Character`. If this function succeeds, it returns a string consisting of `n` copies of `Character` in `Matched` and returns any remaining characters in `Remainder`. `Matched` and `Remainder` are undefined if this function returns **false**.

`@firstnChar( InputStr, Character, n, *Remainder*, *Matched* )`

This function returns true if the first `n` characters of `InputStr` are equal to `Character`. The `n+1st` character may or may not be equal to `Character`. If this function succeeds, it returns a string consisting of `n` copies of `Character` in `Matched` and returns any remaining characters in `Remainder`.

`@nOrLessChar( InputStr, Character, n, *Remainder*, *Matched* )`

This function always returns **true**. It matches up to `n` copies of `Character` at the beginning of `InputStr`. More than `n` characters can be equal to `Character` and this routine will still succeed. However, this routine only matches the first `n` copies of `Character` in `InputStr`. It copies the matched characters to the `Matched` string and copies any remaining characters to the `Remainder` string.

`@nOrMoreChar( InputStr, Character, n, *Remainder*, *Matched* )`

The `normorechar` function matches any string that begins with at least `n` copies of `Character`. If it succeeds, it copies the sequence of `Character` chars to the `Matched` string and copies any remaining characters (that must begin with something other than `Character`) to the `Remainder` string. This function fails and returns **false** if the string doesn't begin with at least `n` copies of *Character*. Note that the `Remainder` and `Matched` variables are undefined if this function fails.

`@ntomChar( InputStr, Character, n, m, *Remainder*, *Matched* )`

This function returns true if the first `n` characters of `InputStr` are equal to *Character*. It will match up to `m` characters (`m >= n`). The `m+1st` character does not have to be different than `Character`, although this function will match, at most, `m` characters. If this function succeeds, it copies the matched characters to the `Matched` string and any following characters to the `Remainder` string. If this function fails and returns **false**, the values of `Matched` and `Remainder` are undefined.

`@exactlyntomChar( InputStr, Character, n, m, *Remainder*, *Matched* )`

This function succeeds and returns **true** if there are at least `n` copies of `Character` at the beginning of `InputStr` and no more than `m` copies of `Character` at the beginning of `InputStr`. If this function succeeds, it copies the matched characters at the beginning of `InputStr` to the `Matched` parameter and any following characters to the `Remainder` parameter. If this function fails, the values of `Remainder` and `Matched` are undefined upon return.

```
@peekiChar
@oneiChar
@uptoiChar
@zeroOrOneiChar
@zeroOrMoreiChar
```

```
@oneOrMoreiChar
@exactlyniChar
@firstniChar
@nOrLessiChar
@nOrMoreiChar
@ntomiChar
@exactlyntomiChar
```

These functions use the same syntax as the standard `xxxxxChar` functions. The difference is that these functions do a case insensitive comparison of the `Character` parameter with the `InputStr` parameter.

`@matchStr( InputStr, String, *Remainder*, *Matched* )`

This function checks to see if the string specified by `String` appears as the first set of characters at the beginning of `InputStr`. This function returns **true** if `InputStr` begins with `String`. If this function succeeds, it copies `String` to `Matched` and any following characters to `Remainder`.

`@matchiStr( InputStr, String, *Remainder*, *Matched* )`

Just like @`matchStr` except this function does a case insensitive comparison.

`@uptoStr( InputStr, String, *Remainder*, *Matched* )`

The `uptoStr` function matches all characters in *InputStr* up to, but not including, the string specified by `String`. If it succeeds, it copies all the matched characters (not including the string specified by `String`) into the `Matched` parameter and any following characters to `Remainder`. If this function returns **false**, the values of `Remainder` and `Matched` are undefined.

`@uptoiStr( InputStr, String, *Remainder*, *Matched* )`

Same as @`uptoStr` function except that this function does a case insensitive comparison.

`@matchToStr( InputStr, String, *Remainder*, *Matched* )`

This function is similar to @`uptoStr` except this function matches all characters up to and including the characters in the `String` parameter.

`@matchToiStr( InputStr, String, *Remainder*, *Matched* )`

Same as @`matchToStr` except this function does a case insensitive comparison.

`@matchID( InputStr, *Remainder*, *Matched* )`

This is a special matching function that matches characters in `InputStr` that correspond to an HLA identifier. That is, `InputStr` must begin with an alphabetic character or an underscore and @`matchID` will match all following alphanumeric or underscore characters. If this function succeeds by matching a prefix of `InputStr` that looks like an identifier, it copies the matched characters to `Matched` and all following characters to `Remainder`. This function returns **false** if the first character of `InputStr` is not an underscore or an alphabetic character. Note that the first character beyond a matched identifier can be anything other than an alphanumeric or underscore character and this function will still succeed.

`@matchIntConst( InputStr, *Remainder*, *Matched* )`

This function matches a string of one or more decimal digit characters (i.e., an unsigned integer constant). The `Matched` parameter, if present, must be an **int32 val** object. If @`matchIntConst` succeeds, it will convert the string to an integer and copy this integer to the `Matched` parameter; it will also copy any characters following the integer string to the `Remainder` parameter.

`@matchRealConst( InputStr, *Remainder*, *Matched* )`

This function matches a sequence of characters at the beginning of `InputStr` that correspond to a real constant (note that a simple sequence of digits, i.e., an integer, satisfies this). The number may have a leading plus or minus sign followed by at least one decimal digit, an optional fractional part and an optional exponent part (see the definition of an HLA real literal constant for more details). If this function succeeds, it converts the string to a **real80** value and stores this value into `Matched` (which must be a **real80 val** object). The characters after the matched string are copied into the `Remainder` parameter. If this function fails, the values of `Matched` and `Remainder` are undefined.

`@matchNumericConst( InputStr, *Remainder*, *Matched* )`

This is a combination of @`matchRealConst` and @`matchIntConst`. It checks the prefix of `InputStr`. If it corresponds to an integer constant it will behave like @`matchIntConst`. If the prefix string corresponds to a real constant, this function behaves like @`matchRealConst`. If the prefix matches neither, this function returns **false**.

`@matchStrConst( InputStr, *Remainder*, *Matched* )`

This function matches a sequence of characters that correspond to an HLA literal string constant. Note that such constants generally contain quotes surrounding the string. If this function returns true, it copies the matched string, minus the quote delimiters, to the `Matched` parameter and it copies the following characters to the `Remainder` parameter. If this function fails, those two parameter values are undefined.

This function automatically handles several idiosyncrasies of HLA literal string constants. For example, if two adjacent quotes appear within a string, @`matchStrConst` copies only a single quote to the `Matched` parameter. If two quoted strings appear at the beginning of `InputStr` separated only by whitespace (a space or any control character other than NUL), then this function concatenates the two strings together. Likewise, any character objects (surrounded by apostrophes or taking the form #ddd, #$hh, or #%bbbbbbbb where ddd is a decimal constant, hh is a hexadecimal constant, and bbbbbbbb is a binary constant) are automatically concatenated into the result string. See the definition of HLA literal constants for more details.

`@zeroOrMoreWS( InputStr, *Remainder* )`

This function always succeeds. It matches zero or more whitespace characters (white space is defined here as a space or any control character other than NUL [ASCII code zero]). This function copies any characters following the white space characters to the `Remainder` parameter (this could be the empty string).

`@oneOrMoreWS( InputStr, *Remainder* )`

This function matches one or more whitespace characters (white space is defined here as a space or any control character other than NUL [ASCII code zero]). If this function succeeds, it copies any characters following the white space characters to the `Remainder` parameter. If this function fails, the `Remainder` string's value is undefined.

`@WSorEOS( InputStr, *Remainder* )`

This function always succeeds. It matches zero or more whitespace characters (white space is defined here as a space or any control character) or the end of string token (a zero terminating byte). This function copies any characters following the white space characters to the `Remainder` parameter (this could be the empty string if it matches EOS or there is only white space at the end of the string).

`@WSthenEOS( InputStr)`

This function matches zero or more whitespace characters (white space is defined here as a space or any control character) immediately followed by the EOS token (a zero terminating byte). Technically, it allows a `Remainder` parameter, but such a parameter will always be set to the empty string if this function succeeds, so it's hardly useful to supply the parameter.

`@peekWS( InputStr, `*`Remainder`*` )`

This function returns true if the first character if `InputStr` is a white space character. If it succeeds and the `Remainder` parameter is present, this function copies `InputStr` to `Remainder`.

`@EOS( InputStr )`

This function returns **true** if `InputStr` is the empty string.

`@reg( InputStr )`

This function returns **true** if `InputStr` matches a valid register name.

`@reg8( InputStr )`

This function returns **true** if `InputStr` matches a valid eight-bit register name.

`@reg16( InputStr )`

This function returns **true** if `InputStr` matches a valid 16-bit register name.

`@reg32( InputStr )`

This function returns **true** if `InputStr` matches a valid 32-bit register name.

13.8.5 Symbol and constant related functions and assembler control functions

`@name( identifier )`

This function returns a string of characters that corresponds to the name of the identifier (note: after text/macro expansion). This is useful inside macros when attempting to determine the name of a macro parameter variable (e.g., for error messages, etc). This function returns the empty string if the parameter is not an identifier.

`@type( `*`identifier_or_expression`*` )`

This function returns a unique integer value that specifies the type of the specified symbol. Unfortunately, this unique integer may be different across assemblies. Do not use this function when comparing types of objects in different source code modules. This is a deprecated function. Future versions of the assembler will return the same value as **@typename**. Do not use this function in new code, and change any existing uses to use **@typename** instead.

`@typename( `*`identifier_or_expression`*` )`

This function returns the string name of the type of the identifier or constant expression. Examples include "int32", "boolean", and "real80".

```
@basetype( identifier_or_expression )
```

Similar to **@typename**, except this function returns the underlying primitive type for array and pointer objects. For other types, it behaves just like **@typename**.

```
@ptype( identifier_or_expression )
```

This function returns a small integer constant denoting the primitive type of the specified identifier or expression. Primitive types would include things like **int32**, **boolean**, and **real80**. See the "hla.hhf" header file for the latest set of constant definitions for **@pType**. At the time this was written, the definitions were (though don't count on these particular values):

```
// pType constants.

hla.ptIllegal:= 0;
hla.ptBoolean:= 1;
hla.ptEnum:= 2;

hla.ptUns8:= 3;
hla.ptUns16:= 4;
hla.ptUns32:= 5;
hla.ptUns64:= 6;
hla.ptUns128:= 7;

hla.ptByte:= 8;
hla.ptWord:= 9;
hla.ptDWord:= 10;
hla.ptQWord:= 11;
hla.ptTByte:= 12;
hla.ptLWord:= 13;

hla.ptInt8:= 14;
hla.ptInt16:= 15;
hla.ptInt32:= 16;
hla.ptInt64:= 17;
hla.ptInt128:= 18;

hla.ptChar:= 19;
hla.ptWChar:= 20;

hla.ptReal32:= 21;
hla.ptReal64:= 22;
hla.ptReal80:= 23;
hla.ptReal128:= 24;

hla.ptString:= 25;
hla.ptZString:= 26;
hla.ptWString:= 27;
hla.ptCset:= 28;

hla.ptArray:= 29;
hla.ptRecord:= 30;
hla.ptUnion:= 31;
hla.ptRegex:= 32;
hla.ptClass:= 33;
hla.ptProcptr:= 34;
hla.ptThunk:= 35;
```

```
        hla.ptPointer:= 36;

        hla.ptLabel:= 37;
        hla.ptProc:= 38;
        hla.ptMethod:= 39;
        hla.ptClassProc:= 40;
        hla.ptClassIter := 41;
        hla.ptIterator:= 42;
        hla.ptProgram:= 43;
        hla.ptMacro:= 44;
        hla.ptText:= 45;
        hla.ptRegExMac:= 46;

        hla.ptNamespace:= 47;
        hla.ptSegment:= 48;
        hla.ptAnonRec:= 49;
        hla.ptAnonUnion := 50;
        hla.ptVariant:= 51;
        hla.ptError:= 52;

        // Total Number of ptypes we support:

        hla.sizePTypes:= 53;
```

`@baseptype( ` *`identifier_or_expression`* ` )`

This function returns a small integer constant denoting the underlying primitive type of the specified identifier or expression. See the discussion for **@ptype** for details. The difference between **@ptype** and **@baseptype** is that **@baseptype** returns the element type for arrays and the base type for *ptPointer* types.

`@class( ` *`identifier_or_expression`* ` )`

This returns a symbol's class type. The class type is constant, value, variable, static, etc., this has little to do with the class abstract data type See the "hla.hhf" header file for the current symbol class definitions. At the time this was written, the definitions were:

```
        hla.cIllegal:= 0;
        hla.cConstant:= 1;
        hla.cValue:= 2;
        hla.cType := 3;
        hla.cVar  := 4;
        hla.cParm := 5;
        hla.cStatic:= 6;
        hla.cLabel:= 7;
        hla.cProc := 8;
        hla.cIterator:= 9;
        hla.cClassProc:= 10;
        hla.cClassIter  := 11;
        hla.cMethod:= 12;
        hla.cMacro:= 13;
        hla.cKeyword:= 14;
        hla.cTerminator:= 15;
        hla.cRegEx:= 16;
        hla.cProgram:= 17;
        hla.cNamespace:= 18;
        hla.cSegment    := 19;
        hla.cRegister:= 20;
        hla.cNone := 21;
```

`@size( identifier_or_expression )`

This function returns the size, in bytes, of the specified object.

`@elementsize( identifier_or_expression )`

This function returns the size, in bytes, of an element of the specified array. If the parameter is not an array identifier, this function generates an assembly-time error.

`@offset( identifier )`

For **var, parm, method**, and class **iterator** objects only, this function returns the integer offset into the activation record (or object record) of the specified symbol.

`@staticname( identifier )`

For **static/readonly/storage** objects, procedures, methods, iterators, and external objects, this function returns a string specifying the "static" name of that string. HLA emits this name to the assembly output file for certain objects (when producing an assembly language output file).

`@lex( identifier )`

This function returns an integer constant specifying the static lexical nesting for the specified symbol. Variables declared in the main program have a lex level of zero. Variables declared in procedures (etc.) that are in the main program have a lex level of one. This function is useful as an index into the *_display_* array when accessing non-local variables.

`@IsExternal( identifier )`

This function returns true if the specified identifier is an external symbol.

`@arity( identifier_or_expression )`

This function returns zero if the specified identifier is not an array. Otherwise, it returns the number of dimensions of that array.

`@dim( array_identifier_or_expression )`

This function returns a single array of integers with one element for each dimension of the array passed as a parameter. Each element of the array returned by this function gives the number of elements in the specified dimension. For example, given the following code:

```
val threeD: int32[ 2, 4, 6];
    tdDims:= @dim( threeD );
```

The `tdDims` constant would be an array with the three elements [2, 4, 6];

`@elements( array_identifier_or_expression )`

This function returns the total number of elements in the specified array. For multi-dimensional array constants, this function returns the number of all elements, not just a particular row or column.

`@defined( identifier )`

This function returns **true** if the specified identifier is has been previously defined in the program and is currently in scope.

`@pclass( identifier )`

If the specified identifier is a parameter, this function returns a small integer indicating how the parameter was passed to the function. These constants are defined in the hla.hhf header file. At this time this document was written, these constants had the following values.

```
hla.illegal_pc:= 0;
hla.valp_pc:= 1;
hla.refp_pc:= 2;
hla.vrp_pc:= 3;
hla.result_pc:= 4;
hla.name_pc:= 5;
hla.lazy_pc:= 6;
```

`valp_pc` means pass by value. `refp_pc` means pass by reference. `vrp_pc` means pass by value/result (value/returned). `result_pc` means pass by result. `name_pc` means pass by name. `lazy_pc` means pass by lazy evaluation.

`@localsyms(` *`record_union_procedure_method_or_iterator_identifier`* `)`

This function returns an array of string listing the local names associated with the argument. If the argument is a record or union object, the elements of the string array contain the field names for the specified record or union. Note that the field names appear in their declaration order (that is, element zero contains the name of the first field, element one contains the name of the second field, etc.).

If the argument is a procedure, method, or iterator, the string array this function returns is a list of all the local identifiers in that program unit. Note that the local object names appear in the reverse order of their declarations (that is, element zero contains the name of the last local name in the program unit, element one contains the second identifier, etc.). Note that parameters are considered local identifiers and will appear in this array. Also note that HLA automatically predefines several symbols when you declare a program unit; those HLA declared symbols also appear in the array of strings **@localsyms** creates.

Currently, **@localsyms** does not allow namespace, program, or class identifiers. This restriction may be lifted in the future if there is sufficient need.

`@isconst(` *`expr`* `)`

This function returns true if the specified parameter is a constant identifier or expression.

`@isreg(` *`expr`* `)`

This function returns **true** if the specified parameter is one of the 80x86 general purpose registers. It returns **false** otherwise.

`@isreg8(` *`expr`* `)`

This function returns **true** if the specified parameter is one of the 80x86 eight-bit general purpose registers. It returns **false** otherwise.

`@isreg16(` *`expr`* `)`

This function returns **true** if the specified parameter is one of the 80x86 16-bit general purpose registers. It returns **false** otherwise.

`@isreg32(` *`expr`* `)`

This function returns **true** if the specified parameter is one of the 80x86 32-bit general purpose registers. It returns **false** otherwise.

`@isfreg( expr )`

This function returns **true** if the specified parameter is one of the 80x86 FPU registers. It returns **false** otherwise.

`@ismem( expr )`

This function returns **true** if the specified expression is a memory address.

`@isclass( expr )`

This function returns **true** if the specified parameter is a class or a class object.

`@istype( identifier )`

This function returns **true** if the specified identifier is a type id.

`@linenumber`

This function returns the current line number in the source file.

`@linenumberstk( expr )`

The expression must be a small unsigned integer value. @linenumberstk(0) returns the current line number in the source file (exactly like @linenumber). If the expression evaluates to some value larger than zero, the @linenumberstack crawls up the macro/include/text expansion/regular expression include stack and prints the line number of the invocation at the level specified by the argument. Note that @linenumberstk(1) prints the line number of the invocation of the current macro (or include, etc.). If the expression is larger than the number of entries on the line number stack, this function returns the line number of the first invocation.

`@filename`

This function returns the name of the current source file.

`@filenamestk( expr )`

The expression must be a small unsigned integer value. @filenamestk(0) returns the current filename for the source file (exactly like @filename). If the expression evaluates to some value larger than zero, the @filenamestack crawls up the macro/include/text expansion/regular expression include stack and prints the filename of the invocation at the level specified by the argument. If the expression is larger than the number of entries on the filename stack, this function returns the filename of the main file.

`@curlex`

This function returns the current static lex level (e.g., zero for the main program).

`@curoffset`

This function returns the current **var** offset within the activation record.

`@curdir`

This function returns +1 if processing parameters, it returns -1 otherwise. This corresponds to whether variable offsets are increasing or decreasing in an activation record during compilation. This function also returns +1 when processing fields in a record or class. This function returns zero when processing fields in a union.

`@addofs1st`

This function returns **true** when processing local variables, it returns **false** when processing parameters and record/class/union fields.

`@lastobject`

This function returns a string containing the name of the last macro object processed.

`@curobject`

This function returns a string containing the name of the last class object processed.

`@curvar`

This function returns a string containing the name of the last memory object processed.

13.8.6 Pseudo-Variables

HLA provides several special identifiers that act as functions in expressions and as variables in **val** assignments. These "pseudo-variables" let you control the code emission during compilation. Typically, you would use these pseudo-variables in a statement like "?@bound:=true;" in order to set their values.

`@errorprefix`

This variable contains a string (default the empty string). Whenever the assembler reports an error message, it first checks this string to see if it is not the empty string. If the string is not the empty string, then HLA will print this string before printing the error message. This pseudo-variable is useful when processing macros and an error message might not appear until deep into the macro expansion. The programmer can set up a helpful string (perhaps using the @lineNumberStack and/or @fileNameStack functions) to print when an error occurs.

`@parmoffset`

This variable contains the starting offset for parameters. This is generally eight for most procedures since the parameters start at offset eight. You can change this value during assembly by assigning a value to this variable (e.g., ?@parmoffset = 10;). However, this activity is not recommended except by advanced programmers.

`@localoffset`

This variable returns the starting offset for local variables in an activation record. This is typically zero. You can change this value during assembly by assigning a value to this variable (e.g., ?@localoffset = -10;). However, this activity is not recommended except by advanced programmers.

`@basereg`

This variable returns a string containing either "ebp" or "esp". You assign either `ebp` or `esp` (the registers, not a string) to this variable. This sets the base register that HLA uses for automatic (**var**) variables. The default is `ebp`. Examples:

```
?SaveBase :string := @basereg;
?@basereg := esp;
<< code that uses esp to access locals and parameters>>
?@basereg := @text( SaveBase ); // Restore to original register.
```

Note the use of `@text` to convert the string to an actual register name. This must be done because HLA only allows the assignment of the actual `ebp/esp` registers to `@basereg`, not a string.

`@enumsize`

This assembly time variable specifies the size (in bytes) of enumerated objects. This has a default value of one.

`@minparmsize`

This assembly time variable has the initial value four. You should not change the value of this object when running under Win32, *NIX, or other 32-bit OS.

`@bound`

This assembly time variable is a boolean value that indicates whether HLA compiles the **bound** instruction into actual machine code (or ignores the **bound** instruction).

`@into`

This assembly time variable is a boolean value that indicates whether HLA compiles the **into** instruction into actual machine code.

```
@label
```

This assembly time variable is an integer value that must be assigned a value greater than zero. This value controls how HLA generates internal unique symbols. HLA normally translates non-external/non-public symbols to some form such as "*originalSymbol__XXX_nnnn*" where originalSymbol is the identifier appearing in the HLA source file, *XXX* is some special string (currently "HLA" as this is being written, but this is subject to change in future versions of HLA), and *nnnn* is a decimal integer string. HLA increments the value of *nnnn* for each symbol it generates, thus ensuring that all internal symbols are unique within a given source file.

A problem can occur with HLA's unique symbol generation algorithm if you're generating an assembly language source file for use with an assembler such as MASM that has an option to make all symbols public. Usually, symbols of the form "*originalSymbol__XXX_nnnn*" are private to a given source file, such symbols are almost never public. However, if you compile HLA code to a MASM source file and them compile the MASM code with the "all symbols public" option, it's quite possible, when linking multiple files together, that you wind up with duplicate symbol errors from the linker. In such (rare) cases, you can use the @label pseudo-variable to work around this problem by changing the value HLA uses for its internal label counter. For example, if the linker complains that the symbol "false__HLA_1023" is multiply defined, you can use the @label pseudo-variable to change the symbol number suffixes in one of the source files using a statement like following near the beginning of your source file:

?@label := 5000;

Be careful about using this pseudo-variable; you should only change the value once and you should only change it near the beginning of a source file. If you reset the @label value to a smaller value somewhere beyond the start of the source file, you can create internal symbol conflicts in your source file. Use this option with care!

```
@exceptions
```

This assembly time variable controls whether HLA emits full exception handling code or an abbreviated set of routines. If this variable contains **true**, then HLA emits the full exception handling code. If **false**, the HLA emits the minimal amount of code to pass exceptions on to Windows or *NIX. Note that this variable only affects code generation in the main **program**, it does not affect the code generation in a **unit**. This variable must be set to true before the **begin** clause associated with the main program if it is to have any effect. Note that including the EXCEPTS.HHF file automatically sets this to true; so you will have to explicitly set it to **false** if you include this file (or some other file that includes EXCEPTS.HHF, like STDLIB.HHF).

```
@optstring
```

By default, HLA *folds* string constants to generate better code. This means that whenever you ask the compiler to emit code for a string constant like "Hello World" the compiler will first check to see if it has already emitted such a string. If so, the compiler uses the reference to the original string constant rather than emitting a second copy of the string; this shortens the size of your program if there are multiple occurrences of the same string in the program. Since string constants generally go into a read-only section of memory, the program cannot accidentally change this unique occurrence. The **@optstrings** pseudo-variable lets you control this optimization. If **@optstrings** is **true** (the default condition), then HLA folds all duplicate string constants; if **@optstrings** is **false**, then HLA emits duplicate strings to the code.

```
@trace
```

This boolean variable controls the emission of "trace" statements by the HLA compiler. This feature is offered in lieu of a decent debugger for tracing through HLA programs. When this variable is false (the default), HLA emits the code you specify. However, if you set this compile-time variable to true, HLA emits the following code before most statements in the program:

```
_traceLine_( filename, linenumber );
```

The filename parameter is a string that specifies the current filename HLA is processing. The *linenumber* parameter is an **uns32** value that specifies the current line number in the file. You are responsible for supplying the "`_traceLine_`" procedure somewhere in your program. Here's a typical implementation:

```
procedure  trace( filename:string; linenumber:uns32 ); @external(
"_traceLine_" );
procedure trace( filename:string; linenumber:uns32 ); @nodisplay;
begin trace;

    pushfd();  // This function must preserve all registers and flags!
    stdout.put( filename, ": #", linenumber, nl );
    popfd();

end trace;
```

As the comments above note, it is your responsibility to preserve all registers and flags in the `_traceLine_` procedure. If you fail to do this, it will corrupt those values in the code that calls `_traceLine_`.

A common operation inside the `_traceLine_` procedure is to display register values. Don't forget that EBP's and ESP's values are modified by this call. Furthermore, if you do any processing whatsoever at all, the flag values will change. To obtain EBP's value prior to the call, fetch the double-word at address [EBP+0]. To obtain ESP's value, take the value of EBP inside `_traceLine_` and subtract 16 from it (EBP, return address, and eight bytes of parameters are on the stack). Obviously if you build `_traceLine_`'s activation record yourself, these values can change. To display the flag values, access the copy of the FLAGs register you pushed on the stack (at offset [EBP-4] in the code above).

In addition to simply displaying values, you can write some very sophisticated debugging routines that let you set breakpoints, watch values, and so on. Someday the HLA Standard Library will include some trace support functions, until then have fun doing whatever you want.

13.8.7 Text emission functions

`@text( str_expr )`

This function replaces itself with the text of the specified parameter. The result is then processed by HLA. E.g.,

```
@text( "mov( 0, eax );" );
```

The above is equivalent to the single move instruction.

`@string(` *`identifier`* `)`

The identifier must be a constant of type text. HLA replaces this item with the string data assigned to the text object.

`@string:`*`identifier`*

The identifier must be a constant of type text. HLA replaces this item with the string data assigned to the text object. Note that this operation is deprecated. HLA now allows **@string**(textVal) to convert a text object to a string value.

`@tostring:`*`identifier`*

Like **@string:**identifier, the identifier must be a constant of type text. Also like **@string:**identifier, HLA replaces this item with the string data assigned to the text object. However, this function also converts `identifier` from a text to a string object.

13.8.8 Miscellaneous Functions

```
@section
```

This function returns a 32-bit bitmap that identifies the current point in the source. Identification is as follows:

```
Bit 0:    Currently processing the CONST section.
Bit 1:    Currently processing the VAL section.
Bit 2:    Currently processing the TYPE section.
Bit 3:    Currently processing the VAR section.
Bit 4:    Currently processing the STATIC section.
Bit 5:    Currently processing the READONLY section.
Bit 6:    Currently processing the STORAGE section.

Bit 12:Currently processing statements in the "main" program.
Bit 13:Currently processing statements in a procedure.
Bit 14:Currently processing statements in a method.
Bit 15:Currently processing statements in an iterator.
Bit 16:Currently processing statements in a #macro.
Bit 17:Currently processing statements in a #keyword macro.
Bit 18:Currently processing statements in a #terminator macro.
Bit 19:Currently processing statements in a thunk.

Bit 23:Currently processing statements in a Unit.
Bit 24:Currently processing statements in a Program.

Bit 25:Currently processing statements in a record.
Bit 26:Currently processing statements in a union.
Bit 27:Currently processing statements in a class.
Bit 28:Currently processing statements in a namespace.
```

This function is useful in macros to determine if a macro expansion is legal at a given point in a program.

13.9 #Text and #endtext Text Collection Directives

The #TEXT and #ENDTEXT directives surround a block of text in an HLA program from which HLA will create an array of string constants. The syntax for these directives is:

```
 #text( identifier )

   << arbitrary lines of text >>

#endtext
```

The `identifier` must either be an undefined symbol or an object declared in the VAL section.

This directive converts each line of text between the **#text** and **#endtext** directives into a string and then builds an array of strings from all this text. After building the array of strings, HLA assigns this array to the identifier symbol. This is a **val** constant array of strings. The **#text..#endtext** directives may appear anywhere in the program where white space is allowed.

Although these directives provide an easy way to initialize a constant array of strings, the real purpose for these directives is to allow the inclusion of Domain Specific Embedded Language (DSEL) text within an HLA program. Presumably, a parser (written with macros, regular expression macros, and the HLA compile-time language) would process the statements between the **#text** and **#endtext** directives.

13.10 #String and #endstring Text Collection Directives

The **#string** and **#endstring** directives surround a block of text in an HLA program from which HLA will create a single string constant. The syntax for these directives is:

```
 #string( identifier )

   << arbitrary lines of text >>

#endstring
```

Either the *identifier* must be an undefined symbol or an object declared in the **val** section.

These directives are similar in principle to the **#text..#endtext** directives except that they produce a single string (including new line characters) holding the entire block of text rather than an array of strings.

Although these directives provide an easy way to initialize a string, the real purpose for these directives is to allow the inclusion of Domain Specific Embedded Language (DSEL) text within an HLA program. Presumably, a parser (written with macros, regular expression macros, and the HLA compile-time language) would process the statements between the **#string** and **#endstring** directives.

13.11 Regular Expression Macros and the @match/@match2 Functions

Regular expression macros contain sequences of pattern-matching statements that you can use to determine if some string takes a particular form. With HLA's regular expression macros and the attendant @match and @match2 functions, you can develop sophisticated language processors inside HLA and specify whatever syntax you like (well, within certain bounds) for those languages.

Technical Note*: although these features are called "regular expression macros", the purists out there will note that "regular expression" is actually a misnomer here. HLA's regular expression macros actually handle a subset of the context-free languages. This language facility is called "regular expression macros" because most programmers, even those not intimately familiar with automata theory, recognize the term and associate "pattern matching" with the term. Hence the use of the term "regular expression" when "context-free grammar" would probably be a better choice. For those of you who aren't intimately familiar with automata theory design, fear not: the context-free languages are a proper superset of the regular languages and you're not being short-changed here. HLA's "regular expression" macros will actually handle all the stuff you can do with a regular expression, and more.*

Before describing the syntax for a regular expression macro, it's probably best to begin by discussing how you use them in a program. This will better motivate you when this document actually discusses the regular expression syntax.

Regular expressions are used for pattern matching.[1] Generally, a regular expression is applied to some string of text and a boolean "success (matched) / failure (no match)" result comes back from the operation. The HLA compile-time function **@match** (and **@match2**) is how you achieve this task. The basic syntax for the **@match**[2] function is the following:

1. Actually, the purists will argue that regular expressions are used for pattern *generation*, not recognition. Because these two facilities are technically equivalent in theoretical computer science, this documentation will ignore this issue and claim that regular expressions are pattern matching devices.

2. For brevity, this document will use @match to imply the use of @match or @match2. The two functions are almost identical in usage other than how they handle whitespace.

```
@match( stringToMatch, RegexMacroName, ReturnsResult, Remainder,
MatchedString )
```

This function returns the boolean result true if the regular expression specified by *RegexMacroName* matches some prefix of the string *stringToMatch*. The remaining three arguments are optional, though if one argument is present then any preceding arguments must also be present.

The optional *ReturnsResult* argument must be an HLA **val** identifier. The **@match** function will store a special **#return** string into this **val** object. We'll look at what a **#return** string is a little later in this documentation. For now, suffice to say that this is the "text" that the regex macro expands into (regex macros do not expand in-place as standard HLA macros do). If this argument is not present and the regex macro produces a **#return** string, then HLA simply throws away the associated string data.

The optional *Remainder* argument must be an HLA **val** identifier. If this argument is present, then the *ReturnsResult* argument must also be present. This argument is identical to the "remainder" arguments of the string matching functions given earlier. When matching *stringToMatch* with *RegexMacroName*, the regex macro might not match the entire string, only a prefix of the string (this is still a successful match). Any remaining characters that are not matched once **@match** exhausts the regular expression are collected and stored into the *Remainder* argument,, if it is present. **@match** will not generate this string if you do not pass the *Remainder* argument (and the string information is simply thrown away at that point).

The optional *MatchedString* argument must be an HLA **val** identifier. If this argument is present, then the *Remainder* and *ReturnsResult* arguments must also be present. This argument is identical to the "matched" arguments of the string matching functions given earlier. If the regular expression macro successfully matches *stringToMatch*, then **@match** will store a copy of the sequence that has been matched into this **val** argument.

Note that if the **@match** function returns false, because *RegexMacroName* failed to match the characters in *stringToMatch*, then **@match** will not disturb the existing values of the *ReturnsResult*, *Remainder*, and *MatchedString* parameters. Therefore, you should only expect those arguments to contain reasonable values if **@match** returns true.

13.11.1 #regex..#endregex

The syntax for a regular expression macro is very similar to a standard macro declaration. Here is the basic form:

```
#regex macroName ( optional_parameter_list ) : optional_locals_list;

   << regex body >>

#endregex
```

The *optional_parameter_list* and *optional_locals_list* items are identical (in syntax) to a macro declaration. The following **#regex** statements demonstrate some of the legal permutations:

```
#regex noParmsOrLocals;
#regex onParmNoLocals( oneParm );
#regex oneLocalNoParms:oneLocal;
#regex variableParms( a, b, c[] );
#regex stringParms( string parms );
```

It's actually a somewhat rare occurrence for a regular expression macro to have parameters. The semantics for parameters (and locals) are different for compiled and precompiled regular expression macros. Therefore, it's a good idea to avoid using parameters unless they are necessary.

The body of a **#regex** macro consists of zero or more regular expression items following by an optional **#return** clause. If the regular expression body is empty, then the regular expression will match the empty string, which means it will match any string appearing in an **@match** function call. The section *Regular Expression Elements* describes the exact syntax for the body of a regular expression macro. The next section describes the optional **#return** clause.

13.11.2 The #return Clause

A **#regex** macro declaration may optionally contain a **#return** clause immediately after the regular expression body (and immediately before the **#endregex** clause). The **#return** clause specifies a string expression to return (via the *ReturnsResult* argument in the **@match** function call). Here is a typical example:

```
#regex newMov;
  <<body for newMov>>
#returns "mov( eax, ebx )"
#endregex
```

Note that an arbitrary HLA string expression is legal after the **#returns** clause, not just a simple literal constant. So you can use the concatenation operation (+) or any other HLA compile-time string functions to build up the **#return** string. Note that there is no semicolon at the end of the string expression. The **#endregex** properly terminates the string expression.

If no **#return** clause is present in a **#regex** macro, then that **#regex** macro returns the empty string as the **#return** string result.

The main purpose for the **#return** clause is to return some text to expand in the invoking code should the **@match** function succeed. Unlike standard macros, you cannot expect to be able to arbitrarily expand text found in a **#regex** macro because you only "invoke" **#regex** macros in an **@match** function call, and those generally appear in a compile-time boolean expression. For example, if the **#regex** macro above directly emitted the **mov** instruction during the invocation of this macro, you'd get syntax errors whenever you made calls like:

```
#if( @match( "Hello World", newMov ))
  .
  .
#endif
```

because HLA would emit the **mov** instruction right into the boolean expression associated with the **#if** statement (which is syntactically incorrect). By putting the **#return** value into a string and returning that string result, the system can defer the expansion of the text until the caller gets to an appropriate context, e.g., (from earlier)

```
#if( @match( "Hello World", newMov, returnResult ))

    @text( returnResult );

#endif
```

This example expands the "mov(eax, ebx)" instruction if and only if the pattern matches "Hello World".

If you would like the default situation to be "expand text if match" then it's easy enough to write a macro to do this job for you:

```
#macro expand( theStr, theRegex ):returnResult;

    #if( @match( theStr, theRegex, returnResult ))

        @text( returnResult );

    #endif

#endmacro
  .
  .
expand( "Hello World", newMov );
```

The return string is automatically processed by the **#match(regex)..#endmatch** block. See the description of **#match..#endmatch** for more details.

13.11.3 Regular Expression Elements

The "meat" of a regular expression macro is the sequence of regular expression elements that appear in a **#regex** macro body. Each element in a regular expression body can match a part of the source string. The following subsections describe each regular expression element in detail.

With only a couple exceptions (that will be noted as they arrive), each time a regular expression element matches a character in the source string (the first parameter provided to **@match**), the match operation *consumes* that character. For example, if the source string is "Hello World" and the first regular expression element matches the single character ‘H’, then ‘H’ is consumed from the source string (yielding "ello World") and further regular expression elements operate on that remainder of the string.

13.11.4 Kleene Star, Plus, and Numeric Range Specifications

Most regular expression elements we’re about to explore match a single instance of themselves. For example, a literal character constant in the body of a regular expression macro will match a single character in the source string (see the next section). You can modify this match operation by supplying one of the following suffixes to the literal character constant.

Suffix	Meaning
*	(Kleene star) Matches zero or more occurrences of the preceding operand.
+	(Kleene plus) Matches one or more occurrences of the preceding operand.
:[n]	Matches exactly *n* occurrences of the preceding operand. *’* must be a reasonably-valued unsigned integer constant expression.
:[n,m]	Matches between *n* and *m* occurrences of the preceding operand. *n* and *m* must be reasonable unsigned integer constants with *n*<m.
:[n,*]	Matches *n* or more occurrences of the preceding operand. *n* must be a reasonably-valued unsigned integer constant expression.

Examples:

```
‘c’*  Matches zero or more ‘c’ characters.
‘c’+  Matches one or more ‘c’ characters.
‘c’:[4] Matches exactly four ‘c’ characters.
‘c’:[4,6] matches between four and six ‘c’ characters.
‘c’:[4,*] Matches four or more ‘c’ characters.
```

Exceptions to this syntax will be noted whenever they occur.

13.11.5 Matching Characters in a Regular Expression

A character literal constant within a **#regex** body matches the corresponding character in the source string. For example, the following regular expression macro matches a string beginning with the single character ‘c’:

```
#regex matchesC;
```

```
    'c'
#endregex
```

Note that this form only allows a single character constant. In particular, you cannot specify an arbitrary HLA character expression. However, you can also use the HLA **@matchChar** (synonym: **@oneChar**) function in a regular expression body to specify a character expression. **@matchChar** requires a single parameter that must evaluate to a single character. For example,

```
#regex matchesC;
    @matchChar( char( uns8('b') + 1)) // Matches 'c'
#endregex
```

The single character match operation consumes a single character from the beginning of the source string if it successfully matches the first character of the source string.

Examples of character matching repetition:

```
'c'*  Matches zero or more 'c' characters.
'c'+  Matches one or more 'c' characters.
'c':[4] Matches exactly four 'c' characters.
'c':[4,6] matches between four and six 'c' characters.
'c':[4,*] Matches four or more 'c' characters.
@matchChar( char( uns8('b') + 1))*  Matches zero or more 'c' characters
```

13.11.6 Case-insensitive Character Matching in a Regular Expression

You can perform a case-insensitive character match by prefixing a literal character constant with the "!" operation. For example, !'c' matches either 'c' or 'C'. Here is an explicit example:

```
#regex matchesCorc;
    !'c'
#endregex
```

If you want to specify a character expression rather than a single literal character constant, you can use the **@matchiChar** function in a manner similar to **@matchChar** given earlier. This operation also consumes a single character from the source string if a match occurs.

Examples of character matching repetition:

```
!'c'*  Matches zero or more 'c' or 'C' characters.
!'c'+  Matches one or more 'c' or 'C' characters.
!'c':[4] Matches exactly four 'c' or 'C' characters.
!'c':[4,6] matches between four and six 'c' or 'C' characters.
!'c':[4,*] Matches four or more 'c' or 'C' characters.
@matchiChar( char( uns8('b') + 1))*  Matches zero or more 'c' or 'C'
characters
```

Note that repetitive matches allow any combination of upper and lower case characters. For example, !'c'+ will match the sequence "ccCcCCc".

13.11.7 Negated Character Matching

Sometimes you'll want to match "anything but a given character." The HLA **#regex** macro body provides a shortcut for matching anything but a single character. By placing a minus sign in front of a single literal character constant, you can tell HLA to match anything but that character. E.g., -'c' matches anything but the 'c' character. You can combine this with the "!" operator to match anything but the upper or lower case version of a character. For example, -!'c' matches anything but 'c' or 'C'.

There is no generic function you can call like **@matchChar** or **@matchiChar** if you want to specify a character expression rather than a character literal constant. However, you can easily achieve the same effect by using negated character sets. See the discussion of matching character sets a little later in this documentation.

If the first character of the source string is not the specified literal constant, then this operation consumes the first character of the source string.

Examples of character matching repetition:

```
-'c'*  Matches zero or more characters that are not 'c'.
-'c'+  Matches one or more characters that are not 'c'.
-'c':[4] Matches exactly four characters that are not 'c'.
-'c':[4,6] matches between four and six characters that are not 'c'.
-'c':[4,*] Matches four or more characters that are not 'c'.
```

13.11.8 String Matching in Regular Expressions

A string literal constant within a **#regex** body matches the corresponding sequence of characters in the source string. For example, the following regular expression macro matches a string beginning with the sequence "str":

```
#regex matchesC;
   "str"
#endregex
```

Note that this form only allows a single literal string constant. In particular, you cannot specify an arbitrary HLA string expression. However, you can also use the HLA **@matchStr** function in a regular expression body to specify a string expression. **@matchStr** requires a single parameter that must evaluate to a single string. For example,

```
#regex matchesHelloWorld;
   @matchStr( "Hello " + "World" ) // Matches "Hello World"
#endregex
```

The string match operation consumes one character from the source string for each character in the regular expression element, but only if the match is completely successful. This is, if the first few characters of the source string match the regular expression element but not all the characters match, then the operation consumes no characters.

Although it is not commonly done, the repetition operations apply to string objects as well as characters. Examples of string matching repetition:

```
"str"*  Matches zero or more "str" sequences.
"str"+  Matches one or more "str" sequences.
"str":[4] Matches exactly four "str" sequences.
"str":[4,6] matches between four and six "str" sequences.
"str":[4,*] Matches four or more "str" sequences.
@matchStr( "Hello" + " world" )*  Matches zero or more "Hello world"
sequences.
```

13.11.9 Case-insenstive String Matching in Regular Expressions

Like character matching, you can do a case-insensitive string match by prefixing a string literal constant with "!" or by using the **@matchiStr** function. E.g.,

```
#regex caseInsensitive;
   @matchiStr( "Hello world" )
#endregex
```

Another example:

```
#regex caseInsensitive;
   !"Hello world"
#endregex
```

Although it is not commonly done, the repetition operations apply to string objects as well as characters. Examples of case-insensitive string matching repetition:

```
!"str"*  Matches zero or more "str" sequences (case insensitive).
!"str"+  Matches one or more "str" sequences (case insensitive).
!"str":[4] Matches exactly four "str" sequences (case insensitive).
!"str":[4,6] matches between four and six "str" sequences (case
insensitive).
!"str":[4,*] Matches four or more "str" sequences (case insensitive).
@matchiStr( "Hello" + " world" )*
   Matches zero or more "Hello world" sequences (case insensitive).
```

13.11.10 Negated String Matching

You can put the "-" operator in front of a string literal expression to specify that the match should fail if the following characters match a given string. For example,

```
#regex caseInsensitive;
   -"Hello world"
#endregex
```

will succeed as long as the next 11 characters are not "Hello world". You can also apply the case-insenstive operator to this sequence,,, e.g., -!"Hello worrld".

Note: negated string matching never consumes any characters from the source string. That is, once this pattern succeeds, the source string contains the same data it did before the match operation. Character consumption doesn't make sense for this operation because the source string could actually be shorter than the negated match string (in which case we still want the pattern to succeed because the source string doesn't begin with the negated string).

The repetition operators to not apply to negated string-matching operations.

13.11.11 String List Matching

The following regular expression syntax tells HLA to successfully match if any one of a list of strings matches the front of the source string:

```
[ "string1", "string2", ..., "stringn" ]
```

The match operation fails only if all the strings in the list fail to match the front of the source string. If multiple strings match the start of the source string, then the first string in the list is the one that will match. So if you want a maximal match, put the longest strings at the beginning of the list, e.g.,

```
[ "these", "the", "th" ]
```

Similarly, if you want a minimal match, put the shortest strings first in the list.

If this operation succeeds, then it consumes the matching characters from the source string.

The repetition operators to not apply to string list matching operations. If you really need this capability, use the alternation operator (discussed later).

13.11.12 Character Set Matching in a Regular Expression

A character set literal constant within a **#regex** body matches a character from the set in the source string. For example, the following regular expression macro matches a string beginning with any of the character ‘c’, ‘s’, or ‘t’:

```
#regex matchesC;
   {‘c’, ‘s’, ‘e’, ‘t’}
#endregex
```

Note that this form only allows a single character set constant. In particular, you cannot specify an arbitrary HLA character set expression. However, you can also use the HLA **@matchCset** (synonym: **@oneCset**) function in a regular expression body to specify a character set expression. **@matchCset** requires a single parameter that must evaluate to a single character. For example,

```
#regex matchesC;
   @matchCset( -{‘c’,’C’} + numericCset ) // Matches anything but ‘c’,
‘C’, or a digit
#endregex
```

The single character set match operation consumes a single character from the beginning of the source string if it successfully matches the first character of the source string.

Examples of character matching repetition:

```
{’0’..’9’}*  Matches zero or more digit characters.
{’0’..’9’}+  Matches one or more digit characters.
{’0’..’9’}:[4] Matches exactly four decimal digit characters.
{’0’..’9’}:[4,6] matches between four and six decimal digit characters.
{’0’..’9’}:[4,*] Matches four or more digit characters.
@matchCset( {"0123456789"})*  Matches zero or more digit characters
```

13.11.13 Negated Character Set Matching

Although you can use the **@matchCset** function to specify a negated character set (e.g., **@matchCset(-someSet)**), for simple literal character set constants HLA allows a shortcut operation. Just put a minus sign in front of the literal character set constant. E.g., -{‘c’, ‘C’,’d’,’D’} matches anything except upper/lower case C and D.

13.11.14 Matching Arbitrary Characters

You can match a single character (regardless of its value) using the negated empty character set (i.e., -{}). However, HLA provides a shortcut for this - the period operator. A period appearing in regular expression body will match any single character and consume that character from the source string. It only fails if there are no more characters in the source string.

```
.*  Matches zero or more characters.
.+  Matches one or more characters.
.:[4] Matches exactly four characters.
.:[4,6] matches between four and six characters.
.:[4,*] Matches four or more characters.
```

The .* pattern is useful at the beginning of a pattern if you want to match some subsequent pattern anywhere in the source string. The .* pattern will skip over any characters up to the desired pattern.

Note that there are some performance issues (at compile time) concerning the use of the repeated "." operator in complex regular expressions. Please see the section on regular expression performance later in this document.

13.11.15 Sequences (Concatenation) - The ‘,’ Operator

Most regular expressions will consist of more than a single regular expression item. The "," operator lets you create a sequence of regular expression items in a regular expression macro. The resulting regular expression is effectively a concatenation of the match semantics. For example, consider the following regular expression macro:

```
#regex identifier;
    {‘a’..’z’, ‘A’..’Z’, ‘_’}, {‘a’..’z’, ‘A’..’Z’, ‘_’}*
#endregex
```

This regular expression matches a sequence of characters that begin with at least one alphabetic or underscore character followed by zero or more alphanumeric or underscore characters (i.e., the definition of an HLA identifier). Here is another example that matches signed integer literal constants:

```
#regex intConst;
    ‘-’:[0,1], {‘0’..’9’}+
#endregex
```

The repetition operators do not apply to sequences (they apply, instead, to the last element of the regular expression sequence). See the discussion of parentheses ("()") for a way to apply a repetition to a sequence.

13.11.16 Alternation - The "|" Operator

The alternation operator ("|") lets HLA select from amongst several different alternative regular expression elements. The basic syntax is:

RX1 | RX2

where RX1 and RX2 are two regular expressions (e.g., the regular expression elements we’ve discussed thus far). The **@match** function will try to match the first regular expression against the source string. If this succeeds, then the whole expression succeeds and the **@match** function ignores the second alternative. If matching the first regular expression fails, then the **@match** function tries to match against the second regular expression. The success or failure of the match is then based on the result of this second match.

Because *R* | *S* is itself a regular expression, recursively we can come up with an arbitrary list of alternatives, e.g.,

```
RX1 | RX2 | RX3 | RX4 | ... | RXn
```

The **@match** function will try to match the first expression. If that fails, it will try the second; if that fails, it will try the third, etc. If any of the *n* regular expressions succeeds, then the alternation succeeds and **@match** ignores any remaining regular expressions in the alternation expression. The alternation sequence fails only if all the subpatterns fail. Note that the string list operator, ["str1", "str2", str3", ..., "strn"] is just a shorthand for:

```
"str1" | "str2" | ... | "strn"
```

The repetition operators do not apply to alternative sequences (they apply, instead, to the last element of the alternation sequence). See the discussion of parentheses ("()") for a way to apply a repetition to an alternation sequence.

13.11.17 Subexpressions - The "()" operator

Like arithmetic operators, regular expression operators exhibit operator precedence. The precedence order is repetitive operators (e.g., "*" and ":[2]"), sequences (","), and last, alternation ("|"). This precedence is natural and eliminates some ambiguity that would otherwise be present in a regular expression. For example, consider the following regular expression sequence:

```
 'c', 'd' | 'e'
```

Does this mean match the string "cd" or "e" (that is, match ‘c’, ‘d’ or match ‘e’), or does this mean match either of the strings "cd" or "cd" (that is, match ‘c’ followed by ‘d’ or ‘e’)? An argument could be made for either resolution of the ambiguity. However, the ‘,’ operator has higher precedence than the "|" operator in HLA, so the first possibility is the one that HLA uses (that is, it matches "cd" or "e").

No matter which choice is made with respect to precedence, there will be situations where you need to override the precedence. As for arithmetic expressions, you can use the parentheses to override precedence. For example, if you really want to match "cd" or "ce" in the previous example, you could rewrite the expression as follows:

```
'c', ( 'd' | 'e' )
```

You may apply the repetition operators to a parenthetical regular expression. For example, the regular expression

```
'c', ( 'd' | 'e' )*
```

matches the character ‘c’ followed by a string of zero or more ‘d’ and ‘e’ characters.

Some regular expression items don’t directly support the repetition operators. For example, sequences don’t support the repetition operators (because of precedence issues). You can use parentheses to overcome this problem, e.g.,

(‘a’, ‘b’, {‘c’,’d’}):+

matches a sequence of characters containing "abc" or "abd" (or both) repeated one or more times.

Note: some operators don’t support repetition because it just doesn’t make sense to do so. Be careful when you force repetition on to an operation that doesn’t otherwise support it. It’s very easy to create a regular expression that never succeeds, or always succeeds, by misapplying the repetition operators.

13.11.18 Extracting Substrings - The Extraction Operator "<>:"

On occasion, you’ll want to save some part of the source string you’ve matched. Granted, the **@match** function has a *MatchedString* argument that returns the entire matched string, but sometimes you'll want to extract only a portion of the entire matched string. The regular expression extraction operator lets you achieve this. The extraction operator uses the following syntax:

```
    < Regular_Expression_sequence >:identifier
```

For the purposes of pattern matching, the extraction operator behaves exactly like the subexpression (parentheses) operator. Everything between the two angle brackets ("<" and ">") is used as a unit. If this sequence matches the source string, then the **@match** function will extract the substring matched by this subexpression and store that string into the compile-time variable specified by *identifier*. This identifier must be a regular expression macro parameter, a regular expression local symbol, or a global **val** object.

One very common use of the **#return** statement is to return some string composed of items processed by the extraction operator. For example, if you want to create a LISP-like assembly language, you could use a regular expression macro like the following (for the **mov, add**, and **sub** instructions):

```
#regex stmt:mnemonic, op1, op2;

    '(',
    <["mov"", "add", "sub"]>:mnemonic, // Match the mnemonic
    ',',
    <.*>:op1, // Everything up to the 2nd comma is the 1st operand
    ',',
    <.*>:op2, // Everything up to the ')' is the 2nd operand
    `)'
#return mnemonic + "(" + op1 + "," + op2 + ")" //Construct HLA statement
#endmacro
```

13.11.19 Invoking Other #regex Macros in a Regular Expression

HLA's **#regex** macros allow you to call other **#regex** macros as though they were pattern matching functions. This one feature alone is what gives HLA's "regular expressions" the power to handle many context-free grammars (rather than being limited to just the regular language subset). If you include the name of some **#regex** macro within a regular expression, the **@match** function will match the current source string using that other regular expression and it's success or failure will determine if the match proceeds upon return from that other **#regex** macro. Consider the following example:

```
#regex ID;
    {'a'..'z', 'A'..'Z', '_'}, {'a'..'z', 'A'..'Z', '0'..'9', '_'}*
#endregex

#regex arrayAccess;
    ID, '[', {'0'..'9'}+, ']'
#endregex
```

The arrayAccess regular expression matches an identifier followed by a numeric constant surrounded by braces, e.g., "myArray[4]".

Regular expression invocations can even be recursive. However, you must be careful not to create an infinitely recursive loop (that is, creating a "left recursive" expression, using compiler terminology). Advanced HLA users (and hopefully you are an advanced HLA user if you're reading this stuff) might think that they can use HLA's conditional assembly directives (e.g., **#if**) to halt the recursion. Though the compile-time language elements may appear in a **#regex** macro, they don't work the way you probably think that they do; in particular, they cannot be used to terminate left recursion. There primary ways to make decisions in regular expressions is via success/failure and via alternation. Specifically, if you have two regular expressions *R* and *S*, then the expression "R, S" will not execute *S* if *R* fails. Similarly, the sequence "R | S" will not execute *S* if *R* succeeds. If these two sequences are inside *S*, then you can stop infinite recursion via the success or failure of *R*.

Eliminating left recursion (and left factoring, another important operation for creating grammars that a predictive parser like **@match** can use) is a subject well beyond the scope of this manual. Pick up any decent compiler design text for details.

There are some important compile-time performance issues associated with invoking regular expression macros from within another regular expression.

13.11.20 Lookahead (peeking)

Sometimes when matching a string, you'll need to look ahead one or more characters to determine whether you can satisfy the current regular expression. A classic example is the "less than" operator in many programming languages ("<"). A simple regular expression of the form '<' is insufficient because the next character might be "=" or ">" (for languages that use "<>" to denote 'not equals', such as HLA). Of course, with HLA's regular expressions you could use the string list ["<=", "<>", "<"] to handle this specific match, but in general you might want the ability to look ahead a character or two before deciding if you're going to succeed. This is accomplished using the peek operator and functions.

For literal constants, prefacing the constant with "/" tells the **@match** function that the following literal constant must appear in the source string, but **@match** will not consume any of those characters. For example, 'a'/'b' requires that the source string begin with "ab" but it only consumes the 'a' from the source string. Similarly, !"ax"/-{'a'..'z', 'A'..'Z', '0'..'9', '_'} matches "ax" (case-insensitive) as long as whatever follows is not an alphanumeric or underscore character (btw, this expression isn't quite good enough, you'll also want to allow end of string after the "ax", but we haven't discussed how to match end of string yet, so that will have to wait).

You can also use the **@peekChar, @peekiChar, @peekStr, @peekiStr,** and **@peekCset** functions to look ahead without consuming any characters in the source string. E.g, this last example is equivalent to:

```
!"ax" @peekCset(-{‘a’..’z’, ‘A’..’Z’, ‘0’..’9’, ‘_’} )
```

13.11.21 Utility Matching Functions

HLA’s regular expression macros support several utility functions that match common strings, thus sparing you from having to write regular expressions for these common items. The following table lists the built-in functions.

Name	Parameters	Supports Repetition	Description
@eos		No	Matches the end of the string.
@ws		Yes	Matches a whitespace character.
@reg		No	Matches an x86 general-purpose 8, 16, or 32-bit register.
@reg8		No	Matches an x86 8-bit register name.
@reg16		No	Matches an x86 16-bit register name.
@reg32		No	Matches an x86 32-bit register name.
@regfpu		No	Matches an x86 FPU register name (HLA syntax: st0, st1, ..., st7).
@regmmx		No	Matches an x86 MMX register name (HLA syntax: mm0, mm1, ..., mm7)
@regxmm		No	Matches an x86 SSE register name (HLA syntax: xmm0, xmm1, ..., xmm7)
@matchid		No	Matches a sequence that looks like an HLA identifier (begins with alphabetic or underscore, followed by zero or more alphanumeric or underscore characters).
@matchIntConst		No	Matches a sequence of one orr more decimal digits.
@matchRealConst		No	Matches a sequence that is a syntactically (HLA) valid floating-point literal constant.
@matchStrConst		No	Matches an HLA string literal (including quotes around the object).

@matchWord	("string")	No	Similar to **@matchStr** (or "literal String") except that the next character after the string it matches must not be alphanumeric or underscore.
@matchiWord	("string")	No	Case-insensitive variant of **@matchWord**.
@arb		Yes	Matches an arbitrary character. Similar to '.' but uses a lazy algorithm rather than a greedy algorithm (that is, it matches as few characters as possible rather than as many characters as possible when the repetition operator allows an arbitrary number of characters).
@pos	(n)	No	*n* is a small unsigned integer. This pattern succeeds if the current character being matched is the n^{th} character in the *original* source string (the one passed to **@match**). Note that the first character in the string is at **@pos(0)**.
@tab	(n)	No	*n* is a small unsigned integer. This pattern succeeds if *n* is greater than or equal to the current character position in the original source string. If the current character position is less than n, then **@tab** matches all characters up to the n^{th} position. Note that the first character in the string is at **@tab(0)**.
@at:*identifier*		No	This function stores the current zero-based index into the source string into the **val** object *identifier* (identifier can also be a **#regex** parameter or local symbol). The type of this value is **uns32**.

13.11.22 Backtracking

#regex regular expressions fully support *backtracking* during pattern matching. This means that if a regular expression ambiguously specifies the text to match (and most non-trivial regular expressions are ambiguous), then the **@match** function will back up and try possible alternatives if one possibility fails. The most obvious example is the alternation operator. If you have a regular expression of the form *R* | *S* and *R* fails to match, then the **@match** function will "back track" in the source string to where *R* began its match ('unconsuming any characters consumed by *R*) and retry the match using *S*.

Alternation certainly isn't the only case where backtracking occurs. Consider the following regular expression:

```
    .*, "hello"
```

This regular expression matches the string "hello" anywhere in the source string. The .* prefix skips over an arbitrary number of characters and then "hello" must match some substring of the source string. Note that the .* regular expression is *greedy*. That is, it will match as many characters as possible. Indeed, when **@match** first encounters .*, it will match the remainder of the string. Such a match, of course, will cause the next patter ("hello") to fail as there are no characters left in the string. When this happens, **@match** will back up some characters (up to the first character that .* matched) and then see if the following regular expression matches. If so, then **@match** succeeds. If **@match** backs up all the way in the source string to where .* began matching in the source string. The **@match** function fails only if it back tracks all the way to the start of what .* matches and then the subsequent pattern still fails.

One thing to note here: because .* is greedy, a regular expression like .*, "hello" will match everything up to the *last* occurrence of "hello" in the source string, not up to the first occurrence. If you would prefer to match up to the first "hello" in the source string, you cannot use a greedy algorithm when skipping arbitrary characters. The **@arb** function matches arbitrary characters, like '.', except it uses a lazy (or deferred) matching algorithm, matching as few characters as possible. An expression like **@arb*** begins by matching zero characters. If the subsequent pattern fails, it matches one character. If the subsequent pattern fails, it tries matching two characters, and so on. Therefore, the regular expression **@arb***, "hello" will match up to the first occurrence of "hello" in the source string.

Backtracking can be a very expensive operation if you're not careful when designing your regular expressions. Consider the following regular expression:

```
'a'+, 'a'+, 'a'+
```

This regular expression (ambiguously) matches three or more 'a' characters. Consider what happens, however, when it is fed a source string such as "aaa". The first 'a'+ term above matches the entire string. This causes the second 'a'+ term to fail, so backtracking occurs. The first 'a'+ term backs off one character and now the second 'a'+ term can succeed. At this point, the third 'a'+ term fails. So the second 'a'+ expression attempts to backtrack, but it fails to match, so the first 'a'+ term backs up one more character. Now, the second 'a'+ term greedily grabs the two available characters. The third 'a'+ term fails at this point, so backtracking occurs yet again. The second 'a'+ term backs up one character and, finally, the third 'a'+ term succeeds. As you can see, this is a lot of work to match a three-character string. In general, backtracking is exponential time complexity (that is, the number of backtracking operations that can take place is proportional to 2**n, where n is the number of regular expression elements). Fortunately, with a little care, you can usually avoid the degenerate cases that exhibit such poor performance. For example, the previous expression could be efficiently written as 'a':[3,*].

Matching an arbitrary number of characters is best done at the end of a regular expression rather than at the beginning or in the middle of a regular expression. Doing so reduces the amount of backtracking that will take place. If you cannot avoid matching an arbitrary sequence of characters, then the next best thing to avoid is having two or more subexpressions in a regular expression that match arbitrary expressions. When you have two or more subexpressions that can match an arbitrary number of characters, backtracking can get ugly. Fortunately, you can usually avoid such degenerate cases by carefully choosing your regular expressions.

13.11.23 Lazy Versus Greedy Evaluation

By default, the algorithms that **@match** uses are greedy. That is, if a given subexpression can match an arbitrary number of characters it will attempt to match as many as possible. If matching too many would cause the match operation to fail, then backtracking will come to the rescue and allow the pattern match to succeed (if at all possible). If all you care about is whether the pattern matches, then it really doesn't matter whether the match algorithm is greedy or non-greedy. There are two cases, however, where you might want to use a non-greedy ("lazy") algorithm: compile-time performance and minimal string matching.

As you saw in the previous section on backtracking, using a greedy algorithm can produce very slow performance in certain degenerate situations. A lazy algorithm (which matches as few characters as possible rather than as many characters as possible) will generally produce much better performance as it can reduce the amount of backtracking that takes place. For example, if you could run the ‘a’+, ‘a’+, ‘a’+ algorithm from the previous section using lazy evaluation, then it would match the first three ‘a’ characters it finds and stop. No backtracking would take place.

Another issue with greedy evaluation is that it always matches the maximum length string. Perhaps this is not what you want. Perhaps you want to match the minimal length string and then process the remainder of the string (after the match) separately. For example, you might expect the following pattern to match everything up to "hello" in the source string and leave the rest of the source string in the remainder operand:

```
.*, "hello"
```

In fact, this regular expression matches everything up to the last occurrence of "hello" in the source string. Therefore, if the source string is something like "hello world, hello people, hello creation" then the remainder string winds up being " creation". Sometimes you want minimal string matching so greedy evaluation is inappropriate.

You can specify lazy evaluation in a pattern using the following repetition forms (assume *R* is some regular expression that supports repetition):

```
R::[n,m]  Matches between n and m copies of R
R::[n,*]  Matches n or more copies of R
```

Although you cannot directly specify lazy evaluation for the unadorned * and + operators, you can easily synthesize lazy evaluation for these operators as follows:

```
R::[0,*]  Matches zero or more copies of R
R::[1,*]  Matches one or more copies of R
```

13.11.24 The @match and @match2 Functions

Consider a simple regular expression that matches a string of the form "id+id" (that is, a simple arithmetic expression). The **#regex** macro might take the following form:

```
#regex simpleExpr;
    @matchID, ‘+’, @matchID
#endregex
```

and you could use this regular expression with an **@match** invocation like this:

```
?boolResult := @match( "value1+value2", simpleExpr );
```

This will work great right up to the point you try something like the following, at which point the pattern matching operation will fail:

```
?falseResult := @match( "value1 + value2", simpleExpr );
```

(notice that there are spaces around the ‘+’ operator in the source string.)

You can solve this problem, and allow arbitrary whitespace in an expression, by inserting **@ws*** regular expressions at appropriate points in your regular expression. For example, you could rewrite *simpleExpr* thusly:

```
#regex simpleExpr;
    @ws*, @matchID, @ws*, ‘+’, @ws*, @matchID
#endregex
```

This new regular expression will ignore whitespace at all the appropriate points in the source string.

There are three problems with sticking **@ws*** terms throughout your regular expression. First, it clutters up the regular expression and makes it difficult to read. Second, it’s easy to misplace (or leave out) one of the **@ws*** terms. Finally, a bunch of terms like **@ws*** can have a serious impact on the processing time needed by **@match** when backtracking occurs.

The **@match2** function solves these three problems. **@match2** automatically skips any white space present before each term it finds in a regular expression that it processes. This spares you having to clutter your code with **@ws*** items, it guarantees that it skips whitespace before each term, and the whitespace it skips is not subject to backtracking issues. Therefore, unless you want absolute control over matching whitespace in your source strings, you should really use the **@match2** function rather than **@match**.

In some very rare cases, you may need the ability to switch between **@match** and **@match2** semantics within the same regular expression. For example, if you want to be able to parse HLA-style character constants, you might be tempted to use a regular expression like the following:

```
"‘’’’" | ‘’’’, ., ‘’’’
```

(That is, match ‘’’’ or a single character surrounded by apostrophes.)

Unfortunately, if you use **@match2** to process this regular expression it will fail when you attempt to match the character constant ‘ ‘. This is because **@match2** will skip the space between the two apostrophes. To avoid this problem, the solution is to make a recursive call to **@match** within the regular expression, as follows:

```
"‘’’’" | @match( ‘’’’, ., ‘’’’ )
```

This guarantees **@match** semantics (no whitespace skipping) for the specified subexpression. Note that there are no returns, remainder, or matched parameters allowed here, and the source string is always the current string being processed.

You can also call **@match2** in a similar manner if you want to guarantee **@match2** semantics in a subexpression.

13.11.25 Compiling and Precompiling Regular Expressions

To improve pattern matching performance, particularly when backtracking occurs, HLA does not interpret the text of a **#regex** macro directly. Instead, HLA compiles a **#regex** macro into an internal format and operates on that internal format rather than on the **#regex** text directly. This effects the operation and usage of **#regex** macros in several subtle ways. To avoid complications when using **#regex** macros, it’s important to understand how compiling **#regex** macros affects their operation.

Prior to the introduction of **#regex** macros, there were two distinct times a programmer had to be concerned with: assembly (compile) time and run time. For example, the **#if** statement operates at compile time whereas the if statement operates at run time. In order to fully utilize the HLA compile-time language, a programmer has to become comfortable with the difference between compile-time operations and run-time code. **#regex** regular expressions also exhibit two distinct phases - compile time and run time - though the confusing part is that both of these phases take place during the HLA compilation phase. Unfortunately, and this is the confusing part, the complete facilities of the HLA compile-time language are only available during regular expression compilation, not while HLA is executing those regular expressions.

Consider, for a moment, the following **#regex** macro definition:

```
#regex sample( count );
   #for( i:= 1 to count )
      ‘a’,
   #endfor
   ‘b’
#endregex
```

At first glance, this code seems rather straightforward. You would think that it would match the number of ‘a’ characters passed as the parameter, followed by a single ‘b’ character. If fact, the behavior is subtlety different. As for machine instructions, the **#for** loop simply replicates the body while compiling the regular expression. Once compiled, the number of matching ‘a’ characters is immutable. For example, if you compile a regular expression using the value 5 as the actual argument value, the above regular expression macro is equivalent to:

```
#regex sample( count );
    'a', 'a', 'a', 'a', 'a',
    'b'
#endregex
```

Unless you recompile this regular expression with a different argument value, the value will never be anything other than five.

Of course, one question that naturally rises is "how does one compile a **#regex** macro?" None of the examples to date have require the use of a special "regular expression compiler" to process a **#regex** macro before using it. Well, as it turns out, HLA will automatically compile a **#regex** macro to its internal form if you use such a macro within an **@match**/**@match**2 function call or if a **#regex** macro name appears within some other regular expression. Because the regular expression is compiled on the spot, the distinction between compile time and run time for the regular expression almost becomes a moot point.

The only problem with compiling a regular expression every time you encounter it is that compilation can be an expensive operation if you recompile a regular expression on each use. Consider the following **#regex** macros:

```
#regex matchHello;
    "hello"
#endregex

#regex hasHello;
    .*, matchHello
#endregex
```

The .* operand in *hasHello* guarantees that backtracking will occur within this regular expression. Unfortunately, on each backtracking instance (and there will be five of them in this case), HLA is forced to recompile the regular expression. This is extremely inefficient. For this reason, you should try to avoid placing uncompiled regular expression macro invocations inside a **#regex** definition. Instead, you should precompile the regular expression to the internal form and specify that compiled version. This saves the expense of recompiling the regular expression on each invocation of the internal **#regex** macro.

The obvious question is "how does one precompile a **#regex** macro?" This is accomplished by creating a **val** object of type **regex** and assigning a **#regex** macro to that **val** identifier. For example:

```
#regex matchHello;
    "hello"
#endregex

val
    compiledMatchHello :regex := matchHello;
```

When HLA sees a statement like this, it compiles the **#regex** macro (*matchHello* in this example) to the internal form and stores this internal data structure into the **regex val** object (*compiledMatchHello* in this example). Now you can use the compiled variant of the **#regex** macro just like the macro itself with one very important difference - compiled regexes do not allow any actual arguments. The processing of the **#regex** parameters (and any HLA compile-time language statements appearing in the macro) takes place when the **#regex** macro is compiled, the statements that would make use of those compile-time language statements is gone when HLA actually executes the regular expression.

If you're only going to use a regular expression macro once in a source file, precompiling the macro won't achieve anything. However, if you use a regular expression macro several times, and especially if you use the regex macro within some other regular expression, you should get in the habit of precompiling the **#regex** macro and using the compiled version. Here's a good convention to use: prefix your **#regex** macro names with an underscore and then immediately follow the **#regex** macro with a **val** statement that compiles the macro to the unadorned name, e.g.,

```
regex _matchHello;
   "hello"
#endregex

val
   matchHello :regex := matchHello;
```

13.11.26 The #match..#endmatch Block

Although you can use **@match** and regular expression macros as generic pattern-matching functions in your HLA compile-time program, the true intended purpose of these pattern-matching facilities is to allow you to write your own "mini-languages" (i.e., domain-specific languages) directly in your HLA source files. The **#match..#endmatch** directives provide a convenient way to compile such domain-specific languages (DSELs). A **#match..#endmatch** block takes the following form:

```
#match( regexID )

   <<body>>

#endmatch
```

The **#match** directive converts the block of text after the closing parenthesis and up to the **#endmatch** directive into a single string, runs **@match** on this string along with the regular expression specified by *regexID*, and then expands the return string to text if the **@match** function returns true. This is roughly equivalent to:

```
?returnStr:string;
#if( @match( <<body text as a string>>, regexID, returnStr ))

   @text( returnStr );

#endif
```

Here is a hypothetical example of **#match..#endmatch** in action:

```
#match( smallBASIClanguage )

   for i = 1 to 10
   print i
   next i

#endmatch
```

Presumably, the *smallBASIClanguage* regular expression would contain the statements to compile the body of the **#match..#endmatch** statement into the corresponding machine instructions.

13.11.27 Using Regular Expressions in Your Assembly Programs

Unless you've had a firm grounding in compiler theory and pattern-matching theory, you're probably wondering what the heck these **#regex** macros are all about. What do they have to do with assembly language? Although this documentation cannot begin to go into details about automata theory and whatnot, it is useful to describe exactly why you might want to create and use **#regex** macros in your assembly programs.

HLA's standard macro facilities let you extend the HLA language, but you don't have a whole lot of say in the design of the syntax for those macro invocations. Though HLA's *context-free macro facilities* provide many options you just don't see in other assemblers, the truth is that you're stuck using the standard HLA syntax when using macros. Regular expressions give you the ability to design a syntax of your own choosing. You can even create full programming languages inside HLA using **#regex** pattern matching macros. All you need to is place your "program" inside some HLA compile-time string object (e.g., using the **#text..#endtext** directive) and then call **@match** to compile your program.

Examples of **#regex** macros appear in the HLA examples download module. Please grab a copy of these examples to see some working examples of HLA **#regex** macros.

13.12The #asm..#endasm and #emit Directives

These directives are deprecated and should not appear in new HLA programs. Much of the need for these statements has gone away over the years as HLA's instruction set was expanded to incorporate most x86 instructions. These statements emit text to an output assembly language source file; obviously, these statements have no effect when HLA produces object code directly.

Probably the biggest use of the #asm..#endasm directive today is to emit comments into the assembly language source file that HLA produces. This is useful if you want to mark a section of the assembly language code to determine statement boundaries in the output code. If you use this scheme to inject comments into the output code, you should always encode your comments as follows:

```
;/* comment text */
```

The ";" character begins comments in all output assembly languages except HLA and Gas; the ';' is a statement separator in HLA and Gas (which is an innocuous output character). The "/*" and "*/" sequences are the comment delimiters in HLA and Gas. Of course, the (pseudo-) HLA output from an HLA compilation is not compilable, so it doesn't really matter if you emit correct comment syntax for pseudo-HLA output, but Gas uses the same comment syntax as HLA so that's the best approach to use if you want your output to be portable across all assemblers.

Note that the HLA back engine will also ignore any text after a ';' up to the end of the line. Therefore, you can emit this text when directly producing object files with the HLABE and it will not impact the output code. Here is an example:

```
program seeCode;
begin seeCode;

    #asm
    ; /* Beginning of main program body */
    #endasm

    mov( 0, eax );
    mov( 1, ebx );
    add( eax, ebx );

    #asm
    ; /* End of main program body */
    #endasm

end seeCode;
```

Here is the code that HLA emits with the "-masm -source" command-line parameters for the main program:

```
_HLAMain proc near32

start   proc near32
start   endp

        call        BuildExcepts__hla_
        pushd       0
        push        ebp
        push        ebp
        lea         ebp, [esp-4]



        ; /* Beginning of main program body */
        mov         eax, 0
        mov         ebx, 1
        add         ebx, eax

        ; /* End of main program body */
QuitMain__hla_::
        pushd       0
        call        dword ptr __imp__ExitProcess@4
_HLAMain endp
```

13.13The #system Directive

The **#system** directive requires a single string parameter. It executes this string as an operating system (shell/command interpreter) operation via the C "system" function call. This call is useful, for example, to run a program during compilation that dynamically creates a text file that an HLA program may include immediately after the **#system** invocation.

Example:

#system("dir")

Note that the **#system** directive is legal anywhere white space is allowable and doesn't require a semicolon at the end of the statement.

13.14The #print and #error Directives

The **#print** directive displays its parameter values during compilation. The basic syntax is the following:

```
#print( comma, separated, list, of, constant, expressions, ... )
```

The **#print** statement is very useful for displaying messages during assembly (e.g., when debugging complex macros or compile-time programs). The items in the **#print** list must evaluate to constant (**const** or **val**) values at compile time.

A common use for **#print** is to display "TODO" messages during compilation, alerting the programmer to features that have yet to be implemented in the application. This helps remind the programmer that code still needs to be written so they don't forget to incorporate that feature. For example,

```
#print( "TODO: Still need to add expression parser here" )
```

The **#error** directive behaves like **#print** insofar as it prints its parameter to the console device during compilation. However, this instruction also generates an HLA error message and does not allow the creation of an object file after compilation. This statement only allows a single string expression as a parameter. If you need to print multiple values of different types, use string concatenation and the **@string** function to achieve this. Example:

```
#error( "Error, unexpected value.  Value = " + #string( theValue ))
```

Notice that neither the **#print** nor the **#error** statements end with a semicolon.

13.15Compile-Time File Output (#openwrite, #append, #write, #closewrite)

These compile-time statements let you do simple file output during compilation. The `#openwrite` statement opens a single file for output, `#write` writes data to that output file, and `#closewrite` closes the file when output is complete. These statements are useful for automatically generating include files that the source file will include later on during the compilation. These statements are also useful for storing bulk data for later retrieval or generating a log during assembly.

The `#openwrite` statement uses the following syntax:

#openwrite(*string_expression*)

This call opens a single output file using the filename specified by the string expression. If the system cannot open the file, HLA emits a compilation error. Note that **#openwrite** only allows one output file to be active at a time. HLA will report an error if you execute **#openwrite** and there is already an output file open. If the file already exists, HLA deletes it prior to opening it (so be careful!). If the file does not already exist, HLA creates a new one with the specified name.

The **`#append`** statement has the same syntax as **`#openwrite`**. The difference is that using **`#append`** will not first delete the file you are opening. Instead, all data written to the file will be appended to the end of the existing file (if any).

The `#write` statement uses the same syntax as the `#print` directive. Note, however, that `#write` doesn't automatically emit a newline after writing all its operands to the file; if you want a newline output you must explicitly supply it as the last parameter to `#write`.

The `#closewrite` statement closes the file opened via `#openwrite` or `#append`. HLA automatically closes this file at the end of assembly if you leave it open. However, you must explicitly close this file before attempting to use the data (via include or `#openread`) in your program. Also, since HLA allows only one open output file at a time, you must use `#closewrite` to close the file before you can open another with `#openwrite`.

Warning: Internally, the `#write` statement simply redirects the standard output stream to send output to the write file and then invokes `#print`, restoring the standard output file handle upon return. This creates a minor problem if there is a syntax error in the `#write` operand list -- the error message is written to the output file! If you're having problems with the `#write` output, temporarily change it to `#print` to see if there's an error in the statement. This defect will probably get fixed in some future version.

13.16Compile-time File Input (#openread, @read, #closeread)

These compile-time statements and function let you do simple file input during compilation. The `#openread` statement opens a single file for input, `@read` is a compile-time function that reads a line of text from the file, and `#closeread` closes the file when input is complete. These statements are useful for reading files produced by `#openwrite/#write/#close`**write** or any other text file during compilation.

The **#openread** statement uses the following syntax:

#openread(*filename*)

The *filename* parameter must be a string expression or HLA reports an error. HLA attempts to open the specified file for reading; HLA prints an error message if it cannot open the file.

The `@read` function uses the following call syntax:

@read(*val_object*)

The *val_object* parameter must either be a symbol you've defined in a `val` section (or via "?") or it must be an undefined symbol (in which case `@read` defines it as a **val** object). `@read` is an HLA compile-time function (hence the "@" prefix rather than "#"; HLA uses "#" for compile-time statements). It returns either **true** or **false**, **true** if the read was successful, **false** if the read operation encountered the end of file. Note that if any other read error occurs, HLA will print an error message and return false as the function result. If the read operation is successful, then HLA stores the string it read (up to 4095 characters) into the `val` object specified by the parameter. Unlike `#openread` and `#closeread`, the **@read** function may not appear arbitrarily in your source file. It must appear within a constant expression since it returns a boolean result (and it is your responsibility to check for EOF).

The `#closeread` statement closes the input file. Since you may only have one open input file at a time, you must close an open input file with `#closeread` prior to opening a second file. Syntax:

```
#closeread
```

Example of using compile-time file I/O:

```
#openwrite( "hw.txt" )
#write( "Hello World", nl )
#closewrite
#openread( "hw.txt" )
?goodread := @read( s );
#closeread
#print( "data read from file = ", s )
```

13.17The Conditional Compilation Statements (#if)

The conditional compilation statements in HLA use the following syntax:

```
#if( constant_boolean_expression )

    << Statements to compile if the >>
    << expression above is true.    >>

#elseif( constant_boolean_expression )

    << Statements to compile if the >>
    << expression immediately above >>
    << is true and the first expres->>
    << sion above is false.         >>

#else

    << Statements to compile if both    >>
    << the expressions above are false. >>

#endif
```

The **#elseif** and **#else** clauses are optional. As you would expect, there may be more than one **#elseif** clause in the same conditional if sequence.

Unlike some other assemblers and high-level languages, HLA's conditional compilation directives are legal anywhere whitespace is legal. You could even embed them in the middle of an instruction! While directly embedding these directives in an instruction isn't recommended (because it would make your code very hard to read), it's nice to know that you can place these directives in a macro and then replace an instruction operand with a macro invocation.

An important thing to note about this directive is that the constant expression in the #IF and #ELSEIF clauses must be of type boolean or HLA will emit an error. Any legal constant expression that produces a boolean result is legal here. In particular, you are limited to expressions like those allowed by the HLA HLL IF statement.

Keep in mind that conditional compilation directives are executed at compile-time, not at run-time. You would not use these directives to (attempt to) make decisions while your program is actually running.

13.18The Compile-Time Loop Statements (#while and #for)

The HLA compile time language also provides a couple of looping structures -- the **#while** loop and the **#for** loop.

The `#while..#endwhile` compile-time loop takes the following form:

```
#while( constant_boolean_expression )

    << Statements to execute as long >>
    << as the expression is true.    >>

#endwhile
```

While processing the `#while..#endwhile` loop, HLA evaluates the constant boolean expression. If it is false, HLA immediately skips to the first statement beyond the `#endwhile` directive.

If the expression is **true**, then HLA proceeds to compile the body of the `#while` loop. Upon encountering the `#endwhile` directive, HLA jumps back up to the `#while` clause in the source code and repeats this process until the expression evaluates false.

Warning: since HLA allows you to create loops in your source code that evaluation during the compilation process, HLA also allows you to create *infinite* loops that will lock up the system during compilation. If HLA seems to have gone off into la-la land during compilation and you're using `#while` loops in your code, it might not be a bad idea to put some `#print` directives into your loop(s) to see if you've created an infinite loop.

Note: because of the limitations of HLA's implementation language (FLEX and BISON), it is not possible to begin a `#while` loop and have the matching `#endwhile` appear in a (different) macro or TEXT constant. When the HLA compiler encounters a `#while` statement it scans the source code looking for the matching `#endwhile` collecting up the statements that make up the body of the loop. During this scan it does not expand TEXT constants or macros. Hence, if you bury the `#endwhile` in a macro or TEXT constant HLA will not be able to find it. For performance and functional reasons, HLA cannot expand macro and TEXT variables during this scan. This is a limitation we will all have to live with until v3.0 of HLA (which will be rewritten in a different language).

The `#for..#endfor` loop can take one of the following forms:

```
#for( loop_control_var := Start_expr to end_expr )

    << Statements to execute as long as the loop control variable's >>
    << value is less than or equal to the ending expression.        >>

#endfor

#for( loop_control_var := Start_expr downto end_expr )

    << Statements to execute as long as the loop control variable's >>
    << value is greater than or equal to the ending expression.     >>

#endfor
```

The HLA compile-time `#for..#endfor` statement is very similar to the for loops found in languages like Pascal and BASIC. This is a definite loop that executes some number of times determine when HLA first encounters the `#for` directive (this can be zero or more times, but the number is computed only once when HLA encounters the `#for`). The loop control variable must be a **val** object or an undefined identifier (in which case, HLA will create a new **val** object with the specified name). In addition, the number control variable must be an eight, sixteen, or thirty-two bit integer value (**uns8, uns16, uns32, int8, int16,** or **int32**). In addition, the starting and ending expressions must be values that an **int32 val** object can hold.

The `#for` loop with the `to` clause initializes the loop control variable with the starting value and repeats the loop as long as the loop control variable's value is less than or equal to the ending expression's value. The `#for..to..#endfor` loop increments the loop control variable on each iteration of the loop.

The `#for` loop with the `downto` clause initializes the loop control variable with the starting value and repeats the loop as long as the loop control variable's value is greater than or equal to the ending expression's value. The `#for..downto..#endfor` loop decrements the loop control variable on each iteration of the loop.

Note that the `#for..to/downto..#endfor` loop only computes the value of the ending expression once, when HLA first encounters the `#for` statement. If the components of this expression would change as a result of the execution of the `#for` loop's body, this will not affect the number of loop iterations.

The `#for..#endfor` loop can also take the following form:

```
#for( loop_control_var in composite_expr )

   << Statements to execute for each element present in the expression >>

#endfor
```

The *composite_expr* in this syntactical form may be a string, a character set, an array, or a record constant.

This particular form of the `#for` loop repeats once for each item that is a member of the composite expression. For strings, the loop repeats once for each character in the string and the loop control variable is set to each successive character in the string. For character sets, the loop repeats for each character that is a member of the set; the loop control variable is assigned the value of each character found in the set (you should assume that the extraction of characters from the set is arbitrary, even though the current implementation extracts them in order of their ASCII codes). For arrays, this `#for` loop variant repeats for each element of the array and assigns each successive array element to the loop control variable. For record constants, the `#for` loop extracts each field and assigns the fields, in turn, to the loop control variable.

Examples:

```
#for( c in "Hello" )
   #print( c )  // Prints the five characters 'H', 'e', ..., 'o'
#endfor

// The following prints a..z and 0..9 (not necessarily in that order):

#for( c in {'a'..'z', '0'..'9'} )
   #print( c )
#endfor

// The following prints 1, 10, 100, 1000

#for( i in [1, 10, 100, 1000] )
   #print( i )
#endfor

// The following prints all the fields of the record type r
// (presumably, r is a record type you've defined elsewhere):

#for( rv in r:[0, 'a', "Hello", 3.14159] )
   #print( rv )
#endfor
```

13.19Compile-Time Functions (macros)

Keep in mind that HLA macros are text expansion devices that may appear anywhere whitespace is allowed. Therefore, you can use them for so much more than 80x86 instruction synthesis. In particular, along with the "?" operator, you can create compile-time functions. For example, consider the following macro that converts the first character of a string to upper case and forces the remaining characters to lower case:

```
program macroFuncDemo;
#include( "stdio.hhf" );


    #macro Capitalize( s );
        @uppercase( @substr( s,0,1), 0 ) +
            @lowercase( @substr( s, 1, 1000 ), 0)
    #endmacro

static
    Hello: string := Capitalize( "hELLO" );
    World: string := Capitalize( "world" );

begin macroFuncDemo;

    stdout.put( Hello, " ", World, nl );

end macroFuncDemo;
```

13.20Sample Macro: A Modified IF..ELSE..ENDIF Statement

In this section we'll create a new kind of IF statement that doesn't nest the same way standard IF statements nest. In particular, if we define the statement such that all IF clauses nested with an outer IF..ENDIF block share the same ELSE and ENDIF clauses. If this were the case, then you could implement some as follows:

```
if( expr1 ) then

   << some 'true' statements >>

   if( expr2 ) then

   << 'true' statements >>


else

   << 'false' statements >>

endif;
```

If *expr1* is false, control immediately transfers to the ELSE clause. If the value of *expr1* is true, the control falls through to the next IF statement.

If *expr2* evaluates false, then the program jumps to the single ELSE clause that all IFs share in this statement. Notice that a single ELSE clause (and corresponding ’false’ statements) appear in this code; hence the code does not necessarily expand in size. If expr2 evaluates true, then control falls through to the ’true’ statements, exactly like a standard IF statement.

Notice that the nested IF statement above does not have a corresponding ENDIF. Like the ELSE clause, all nested IFs in this structure share the same ENDIF. Syntactically, there is no need to end the nested IF statement; the end of the THEN section ends with the ELSE clause, just as the outer IF statement’s THEN block ends.

Of course, we won't actually define a new macro named "if" because if we did (e.g., by using the **#id** statement) we would no longer be able to use standard IF statements in an HLA program (at least, not without the '~' prefix). Further, doing so would make your programs very difficult to comprehend if the IF keyword had different semantics in different parts of the program. The following program uses the identifiers "_if", "_then", "_else", and "_endif" instead. It is questionable if these are good identifiers in production code (perhaps something a little more different would be appropriate). The following code example uses these particular identifiers so you can easily correlate them with the corresponding high-level statements.

```
/***********************************************/
/*                                             */
/* if.hla                                      */
/*                                             */
/* This program demonstrates a modification of */
/* the IF..ELSE..ENDIF statement using HLA's   */
/* multi-part macros.                          */
/*                                             */
/***********************************************/


program newIF;
#include( "stdlib.hhf" )



// Macro implementation of new form of if..then..else..endif.
//
// In this version, all nested IF statements transfer control
// to the same ELSE clause if any one of them have a false
// boolean expression.  Syntax:
//
//  _if( expression ) _then
//
//      <<statements including nested _if clauses>>
//
//  _else // this is optional
//
//      <<statements, but _if clauses are not allowed here>>
//
//  _endif
//
//
// Note that nested _if clauses do not have a corresponding
// _endif clause.  This is because the single _else and/or
// _endif clauses terminate all the nested _if clauses
// including the first one.  Of course, once the code
// encounters an _endif another _if statement may begin.
// Macro to handle the main "_if" clause.
// This code just tests the expression and jumps to the _else
// clause if the expression evaluates false.

macro _if( ifExpr ):elseLbl, hasElse, ifDone;

    ?hasElse := false;
    jf(ifExpr) elseLbl;


// Just ignore the _then keyword.

keyword _then;


// Nested _if clause (yes, HLA lets you replace the main
// macro name with a keyword macro).  Identical to the
// above _if implementation except this one does not
// require a matching _endif clause.  The single _endif
// (matching the first _if clause) terminates all nested
// _if clauses as well as the main _if clause.

keyword _if( nestedIfExpr );
    jf( nestedIfExpr ) elseLbl;

    // If this appears within the _else section, report
    // an error (we don't allow _if clauses nested in
    // the else section, that would create a loop).

    #if( hasElse )

        #error( "All _if clauses must appear before the _else clause" )

    #endif


// Handle the _else clause here.  All we need to is check to
// see if this is the only _else clause and then emit the
// jmp over the else section and output the elseLbl target.

keyword _else;
    #if( hasElse )

        #error( "Only one _else clause is legal per _if.._endif" )

    #else

        // Set hasElse true so we know that we've seen an _else
        // clause in this statement.

        ?hasElse := true;
        jmp ifDone;
        elseLbl:

    #endif
```

```
// _endif has two tasks.  First, it outputs the "ifDone" label
// that _else uses as the target of its jump to skip over the
// else section.  Second, if there was no else section, this
// code must emit the "elseLbl" label so that the false conditional(s)

// in the _if clause(s) have a legal target label.

terminator _endif;

    ifDone:
    #if( !hasElse )

        elseLbl:

    #endif

endmacro;


static
    tr:boolean := true;
    f:boolean := false;

begin newIF;

    // Real quick demo of the _if statement:

    _if( tr ) _then

        _if( tr ) _then
        _if( f ) _then

            stdout.put( "error" nl );

    _else

        stdout.put( "Success" );

    _endif

end newIF;
```

Just in case you're wondering, this program prints "Success" and then quits. This is because the nested "_if" statements are equivalent to the expression "true && true && false" which, of course, is false. Therefore, the "_else" portion of this code should execute.

The only surprise in this macro is the fact that it redefines the *_if* macro as a keyword macro upon invocation of the main *_if* macro. The reason this code does this is so that any nested *_if* clauses do not require a corresponding *_endif* and don't support an *_else* clause.

Implementing an ELSEIF clause introduces some difficulties, hence its absence in this example. The design and implementation of an ELSEIF clause is left to the more serious reader[1].

1. I.e., I don't even want to have to think about this problem!

13.21 Text Processing, Lexical Analysis and the #text..#endtext Block

Although HLA's multi-part macros are very powerful and flexible, they to have some important limitations if you're trying to create your own statements. In particular, if the statements you want to create require some operands, the multi-part macro invocation forces you to specify those operands within parentheses immediately after the macro's name. While you can probably live with this most of the time, there are some situations where you might want to specify the new language feature using a different syntax. Well, with a bit of work it is certainly possible to do this. HLA's compile-time language actually provides all the tools you need to write a full-fledged compiler. While extending HLA in this fashion is well beyond the scope of this text, it is worthwhile to point you in the right direction, just in case you're dying to do really fancy things with HLA.

The key to creating your own personal structures in HLA lies with the HLA compile-time string and pattern matching functions. These functions let you process strings of data in very complex ways, translating that string data into whatever you please. Combined with HLA's **#text..#endtext** blocks, which let you copy a portion of your source file into string variables, it is possible to write an HLA compile-time program that processes those portions of your source files. Of course, once you process your source file as string data, you can use any syntax you choose (and support) within that string data. You can design very sophisticated DSELs using this technique.

The **#text..#endtext** block uses the following syntax:

```
#text( identifier )

    << arbitrary lines of text >>

#endtext
```

The *identifier* symbol must be undefined or a **val** object within the current scope. HLA creates a **val** objected named identifier that will be an array of strings. Each string in the array will contain one line of text between the **#text** and **#endtext** reserved words. The array of strings will contain the text immediately following "#text(identifier)" up to the character just before the **#endtext** directive.

```
// textDemo.hla
//
// This program demonstrates how the #text and #endtext
// directives operate.


program textDemo;


// A quick demonstration of the #text..#endtext directives:

#text( lines ) Hello
World
how are
you #endtext


// Print out the strings gathered into "lines" above
// so you can see the effect of the #text..#endtext directive:

?i := 0;
#while( i < @elements( lines ))

     #print( i, ": '", lines[i], "'" )
```

```
    ?i := i + 1;

#endwhile
#print( "------" )




// A cleaner example (typical of what you would find in DSELs):

#text( MyDSELsource )

    if( x=y && a<b || c<>d ) then

        print "This is my own special language, a=", a;

    endif;

#endtext

// Print the above text (to attempt to actually compile
// those statements in this example!)

?i := 0;
#while( i < @elements( MyDSELsource ))

    #print( i, ": '", MyDSELsource[i], "'" )
    ?i := i + 1;

#endwhile



begin textDemo;
end textDemo;
```

Demonstration of the #TEXT..#ENDTEXT Directives

The program above prints the following when you compile this program with HLA:

```
0: ' Hello'
1: 'World'
2: 'how are'
3: 'you '
------
0: ''
1: ''
2: '    if( x=y && a<b || c<>d ) then'
3: '    '
4: '        print "This is my own special language, a=", a;'
5: '        '
6: '    endif;'
7: '    '
```

```
8: ''
```

As this example suggests, if you want to create a DSEL (Domain Specific Embedded Language) that supports an arbitrary syntax, you would insert your DSEL statements between the **#text** and **#endtext** directives and then use the HLA compile-time language to process this text in the associated array of strings.

In order to process these statements, one of the first activities will be to break up the text into its constituent parts. In the second example above, this would correspond to breaking up those nine strings into:

```
if
(
x
=
y
&&
a
<
b
||
c
<>
d
)
then
print
"This is my own special language, a="
,
a
;
endif
;
```

Each of these pieces is called a *lexeme*. Compiler writers call the process of breaking a stream of text up into lexemes *lexical analysis* or *scanning*. A *lexical analyzer* or *scanner* is the code responsible for actually breaking up the text. While a full treatment of lexical analysis is, again, beyond the scope of this document[1], some simple techniques you can use to write a *scanner* are easy to understand and well within the scope of this chapter.

Some languages ignore white space and new lines in the source code; others treat these characters as part of the syntax. For example, a language such as HLA ignores new lines, you can cram your whole program onto a single physical source code line if you so desire[2]. Traditional assemblers, on the other hand, only allow one statement per line and use the new line sequence to separate these statements. In our current example (MyDSELsource), we'll assume that the language ignores white space and new line characters.

Actually, HLA's **#text..#endtext** block automatically eliminates all new lines appearing in the text. Instead of new lines, HLA copies each line of text (sans new line) to a separate string in the string array. For our example this is unfortunate because it would be more convenient to treat the entire block of text as a single string of characters. (Note: you could also use HLA's **#string..#endstring** block to capture all the text into a single string, complete with newline characters; if you do that, then you get to ignore the **#while** loop below; this example uses **#text..#endtext** to demonstrate the process of processing one line at a time.) Therefore, one of the first jobs of the scanner we are going to write is to combine these separate lines of text back together. One simple solution is to execute some (compile-time) code like the following before attempting to process the text:

1. That subject belongs in a text on compiler design and implementation.
2. That would be really bad programming style, but it is legal syntactically.

```
?i := 0;
?source := "";
#while( i < @elements( MyDSELsource ))

    ?source := source + " " + MyDSELsource[i];
    ?i := i + 1;

#endwhile
```

(Inserting a space between lines is necessary since HLA has removed the original separating new line character sequence. This prevents the end of one line from running directly into the beginning of the next line.)

There are two problems with the code above; first, and least important, is that this code wastes a lot of memory. Once you are done there will be two copies of the source file hanging around in memory. This is especially problematic if there is a lot of text between the **#text** and **#endtext** directives. The second problem with this sequence is that it is slow, especially if it has to process a lot of text.

A better solution is to grab a new line of text only after the scanner has finished processing all the previous text. This is easily handled by including the following compile-time statements at the beginning of the scanner code:

```
// Before executing the following code, you must initialize
// CurrentInput and lineNumber as follows:
//
//    ?lineNumber := 0;
//    ?CurrentInput := MyDSELsource[ 0 ];

?CurrentInput := @trim( CurrentInput, 0 );  // Remove leading spaces from
input.
#while( @length( CurrentInput ) = 0 )

    ?lineNumber := lineNumber + 1;
    #if( lineNumber < @elements( MyDSELsource ) )

        ?CurrentInput := @trim( MyDSELsource[ lineNumber ], 0 );

    #endif

#endwhile
```

Notice that this code only returns an empty string when it exhausts all the lines of text in the **#text**..**#endtext** block. You may test for "end of file" (or, at least, end of this sequence) by explicitly testing for an empty string after the code above executes. Also, note that this code automatically removes any leading and trailing spaces from the text it processes (the call to **@TRIM** handles this). Therefore, when the above code executes, the first item to process appears in the first character of the *CurrentInput* string (assuming, of course, that *CurrentInput* is not empty).

Extracting single character lexemes from the input string is easy. You can use the **@OneChar** function to see if the first character of a string matches a particular character. For example, if the plus and minus signs are special lexemes in your language, then you can use code like the following to see if *CurrentInput* (from above) begins with one of these characters:

```
#if( @OneChar( CurrentInput, '+', CurrentInput ))

    << CurrentInput began with a '+'.  Note that we've extracted
        the '+' from the beginning of CurrentInput in the call above >>

#elseif( @OneChar( CurrentInput, '-', CurrentInput ))
```

```
    << CurrentInput began with a '-'.  Otherwise this is the same
       as the above. >>

#else ...
```

The compile-time pattern matching functions (e.g., **@OneChar**) only store the remainder characters into the remainder operand (the third parameter above, which is *CurrentInput*) if they return true. Therefore, if **@OneChar** in the first **#if** above does not match a plus sign at the beginning of the *CurrentInput* string, it will not change the value of *CurrentInput*; instead, the **#elseif** clause will test the original string. On the other hand, if the first call to **@OneChar** above discovers that *CurrentInput* does begin with a plus sign, then it stores the characters in *CurrentInput* following the plus sign into the remainder operand (which is *CurrentInput*). This deletes the plus sign from the beginning of the string.

To match specific multi-character lexemes, you would use the compile-time **@MatchStr** function. For example, to match the "&&" lexeme, you would use **@MatchStr** as follows:

```
#if( @MatchStr( CurrentInput, "&&", CurrentInput ))

    << Drop down here if CurrentInput begins with "&&" >>
    << (this also extracts "&&" from the string. >>

#else ...
```

Like the **@OneChar** function, the **@MatchStr** call above only deletes the "&&" characters from *CurrentInput* if the string begins with these two characters; otherwise **@MatchStr** does not affect the string.

Extracting single character lexemes is generally quite easy, but you must be careful if some multi-character lexemes begin with the same character as a single character lexeme. For example, "<" is a common single-character lexeme that generally means "less than." Matching "<" as a single character lexeme may create problems if you also need to match the two character lexeme "<=" in your language. If you use the **@OneChar** function as we did above for plus and minus then your code may treat the less than or equal operator as two one-character lexemes rather than as a single two-character lexeme. The solution is to check for the longer lexemes first:

```
#if( @MatchStr( CurrentInput, "<=", CurrentInput ))

    << Come here on "<=" >>

#elseif( @MatchStr( CurrentInput, "<>", CurrentInput ))

    << Drop down here if the string begins with "<>" >>

#elseif( @OneChar( CurrentInput, '<', CurrentInput ))

    << Do this if string begins with '<' but not "<=" or "<>" >>

#else ...
```

Simple lexemes like operators are very easy to process using HLA functions like **@OneChar** and **@MatchStr**. However, there are many string patterns you will want to recognize that do not consist of simple strings. Two common examples are numeric values and identifiers. To recognize these lexemes we must use a general pattern that matches more than a single string. Fortunately, HLA's compile-time pattern matching functions are up to the task.

Let's consider the example of an unsigned decimal integer constant first. Such lexemes begin with a single numeric digit and may contain zero or more additional numeric digits. So the string is always at least one character long and may be longer, as necessary. Recognizing a character from a set of characters is easy; all you need do is call the **@OneOrMoreCset** function to match the value. The following sample code demonstrates how easy this is:

```
#if( @OneOrMoreCset( CurrentInput, {'0'..'9'}, CurrentInput, theNumber ))
```

```
    << At this point, we’ve matched a string of digits,
       "theNumber" contains the string we’ve matched and
       "CurrentInput" contains the remainder of the string >>

#else ... // It wasn’t a numeric lexeme.
```

Note that the call to **@OneOrMoreCset** in the example above supplies the fourth, optional, parameter. If **@OneOrMoreCset** successfully matches a string of digits, it will copy the string it matched into this fourth parameter (which must be a VAL object). This fourth parameter was not necessary in the previous examples because the code knew what string it matched since there was only one possible string it could match. However, the call to **@OneOrMoreCset** above can match a nearly infinite variety of different strings. Since you might actually want to use that value while processing the statements in your language, it’s a good idea to save that value for future use, hence the last parameter above. As usual, if @OneOrMoreCset fails to match the pattern you specify, it does not affect the values of *CurrentInput* or *theNumber*.

Another common pattern you will often need to recognize is a string that represents an identifier. Different languages may specify identifiers differently, but a common definition is that an identifier must begin with an underscore or an alphabetic character and may contain additional alphanumeric or underscore characters (this matches HLA’s definition of an identifier). HLA actually has a special pattern matching function, **@MatchID**, that matches HLA-style identifiers; we will not employ that function here so you can see how to write more complex patterns.

Recognizing an HLA identifier requires two steps: first, we must ensure that the identifier begins with an alphabetic character or an underscore. This is easily accomplished with the following **@PeekCset** function call:

```
      @PeekCset( CurrentInput, {’a’..’z’, ’A’..’Z’, ’_’} )
```

This call to the **@PeekCset** function returns true if *CurrentInput* begins with an underscore or an alphabetic character, it returns false otherwise. It does not affect *CurrentInput’s* value. Therefore, we can use this function to determine if our identifier begins with an appropriate character. Once we know that it begins with an underscore or alphabetic character, we can easily match the entire identifier by calling **@OneOrMoreCset** as follows:

```
@OneOrMoreCset
(
    CurrentInput,
    {’a’..’z’, ’A’..’Z’, ’0’..’9’, ’_’},
    CurrentInput,
    theID
)
```

This call will match all the characters in an identifier and leave those characters in the *theID* string; as usual, it removes the identifier from the beginning of the *CurrentInput* string. You would typically match an identifier using code like the following:

```
#if( @PeekCset( CurrentInput, {’a’..’z’, ’A’..’Z’, ’_’} ))

    #if
    (
        @OneOrMoreCset
        (
            CurrentInput,
            {’a’..’z’, ’A’..’Z’, ’0’..’9’, ’_’},
            CurrentInput,
            theID
        )
    )

        << Okay, we’ve got an identifier and it’s in "theID" >>
    #endif

#else ...
```

If you carefully study the above logic, you might think that you can shorten this to the following code:

```
#if
(
        @PeekCset( CurrentInput, {'a'..'z', 'A'..'Z', '_'} )
    &&  @OneOrMoreCset
        (
            CurrentInput,
            {'a'..'z', 'A'..'Z', '0'..'9', '_'},
            CurrentInput,
            theID
        )
)

        << Okay, we've got an identifier and it's in "theID" >>

#else ...
```

However, there is a subtle flaw in this logic. The HLA compile-time language uses complete boolean evaluation. Therefore, if the call to **@PeekCset** returns false the code above will go ahead and call **@OneOrMoreCset**. Most of the time, this will not adversely affect anything. However, if the next set of characters in the input stream happen to be a set of numeric digits, the call to **@OneOrMoreCset** will return true. Of course, **false** AND **true** is still **false**, but don't forget that **@OneOrMoreCset** has the side effect of modifying *CurrentInput*. This is probably not what you've intended to do. If you are intent on using the "&&" operator, you can use code like to following to eliminate the problem with the side effect that the pattern matching functions will produce:

```
#if
(
        @PeekCset( CurrentInput, {'a'..'z', 'A'..'Z', '_'} )
    &&  @OneOrMoreCset
        (
            CurrentInput,
            {'a'..'z', 'A'..'Z', '0'..'9', '_'},
            Remainder,
            theID
        )
)

        << Okay, we've got an identifier and it's in "theID" >>

        ?CurrentInput := Remainder;

#else ...
```

In this example, the remainder of the string is copied into a temporary variable. The code only overwrites *CurrentInput* (with the temporary value) if the full expression evaluates true.

Most languages will have a set of *reserved words*. Reserved words (or *keywords*) are generally nothing more than identifiers that have special meaning within the context of a language. In the MyDSEL example earlier, it is a good bet that the identifiers *if, then, print,* and *endif* are all reserved words in this DSEL. The easiest way to handle (a small number of) reserved words is to first recognize them as identifiers and then use a sequence of string comparisons to see if the identifier you've matched is actually a reserved word. You could use code like the following to do this:

```
#if( @PeekCset( CurrentInput, {'a'..'z', 'A'..'Z', '_'} ))

    #if
    (
        @OneOrMoreCset
        (
            CurrentInput,
            {'a'..'z', 'A'..'Z', '0'..'9', '_'},
            CurrentInput,
            theID
        )
    )

        #if( theID = "if" )

            << It's the "IF" reserved word >>

        #elseif( theID = "then" )

            << It's the "THEN" reserved word >>

        #elseif( theID = "endif" )

            << It's the "ENDIF" reserved word >>

        #elseif( theID = "print" )

            << It's the "THEN" reserved word >>

        #else

            << Okay, we've got an identifier and it's in "theID" >>

        #endif

    #endif

#else ...
```

HLA lets you design and implement your own complex patterns. However, HLA does contain some built-in pattern matching functions for some common patterns. These include functions that match identifiers (**@MatchID**), integer constants (**@MatchIntConst**), floating-point constants (**@MatchRealConst**), numeric (integer or floating point) constants (**@MatchNumericConst**), and string constants (**@MatchStrConst**). These functions are generally much more convenient to use and certainly more efficient than using patterns you've written to match these types of strings. As long as HLA's idea of an identifier, number, or string is suitable for your application, you should use these pattern-matching functions for these purposes.

In addition to the specialized pattern matching functions above, HLA also provides special pattern matching function that deal with whitespace and the end of a string. These functions include **@ZeroOrMoreWS, @OneOrMoreWS, @WSorEOS, @WSthenEOS, @PeekWS,** and **@EOS**. See the HLA Compile-time Language document for more details on these functions.

With the basic tools and techniques out of the way, now it's time to look at how we would actually write a scanner using the HLA macro (compile-time function/procedure) facilities. The following program provides a small lexer for the "MyDSELsource" example above.

```
// textDemo2.hla
//
// This program demonstrates how to write a lexical
// analyzer (scanner) with the HLA compile-time language.


program textDemo2;



// DSEL text to scan:

#text( MyDSELsource )

   if( x=y && a<b || c<>d ) then

      print "This is my own special language, a=", a;

   endif;

#endtext



// Compile-time function that scans the text above.

macro lexer( Input, index ):CurrentInput, Matched;

   ?CurrentInput:string := "";
   ?Matched:string := "";

   #if( @elements( Input ) = 0 )

      "Expected an array of strings as 'lexer' argument"

   #else

      ?CurrentInput := @trim( Input[ index ], 0 );

      // The following #while loop removes all blank lines.

      #while( @length( CurrentInput ) = 0 && index < @elements( Input ))

         ?index := index + 1;
         #if( index < @elements( Input ))

            ?CurrentInput := @trim( Input[ index ], 0 );

         #else

            ?CurrentInput := "#endtext";

         #endif

      #endwhile
```

```
// If we reached the end of the input, just return
// "#endtext" for this example.  The demo code that
// calls this function automatically stops after this
// point.

#if( index >= @elements( Input ))

    "#endtext"

#else

    // Okay, we've got a non-empty string.
    // Do the lexical analysis on it.

    #if( @OneChar( CurrentInput, '=', CurrentInput ))

        "="  // Return this item as the lexeme.


    // Note: we must check for "<>" before checking for "<".

    #elseif( @MatchStr( CurrentInput, "<>", CurrentInput ))

        "<>"

    #elseif( @OneChar( CurrentInput, '<', CurrentInput ))

        "<"

    #elseif( @OneChar( CurrentInput, '(', CurrentInput ))

        "("

    #elseif( @OneChar( CurrentInput, ')', CurrentInput ))

        ")"

    #elseif( @OneChar( CurrentInput, ',', CurrentInput ))

        ","

    #elseif( @OneChar( CurrentInput, ';', CurrentInput ))

        ";"

    #elseif( @MatchStr( CurrentInput, "&&", CurrentInput ))

        "&&"

    #elseif( @MatchStr( CurrentInput, "||", CurrentInput ))

        "||"

    #elseif( @MatchStrConst( CurrentInput, CurrentInput, Matched ))

        // For the purposes of this program, put the quotes
```

```
                // back around the string constant (@MatchStrConst
                // removes the delimiting quotes).

                ("""" + Matched + """")

            #elseif( @MatchID( CurrentInput, CurrentInput, Matched ))

                // We've matched an ID, see if it is actually one
                // of the reserved words:

                #if( Matched = "if" )

                    ("rw: if")

                #elseif( Matched = "then" )

                    ("rw: then")

                #elseif( Matched = "endif" )

                    ("rw: endif")

                #else

                    // If it's not one of our reserved words, then
                    // just return the ID:

                    ("id: " + Matched)

                #endif

            #else


                #error( "Unexpected lexeme: " + CurrentInput )
                ?CurrentInput := "";
                ""

            #endif
            ?Input[ index ] := CurrentInput;

        #endif

    #endif
    ?CurrentInput:string := "";
    ?Matched:string := "";

endmacro;


val
    lineNumber := 0;


#while( lineNumber < @elements( MyDSELsource))

    #print( lexer( MyDSELsource, lineNumber ))
```

```
#endwhile

begin textDemo2;
end textDemo2;
```

Sample Lexical Analyzer