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

Although HLA's multi-part macros are very powerful and flexible, they do have some important limitations if you're trying to create your own statements with their own, unique, syntax. 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 object 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;

```

Program 1 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 document¹, 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 like HLA ignores new lines, you can cram your whole program onto a single physical source code line if you so desire². 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.

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.

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. 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:

```
?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 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 that you can compare against. 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 fol-

lowing 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 take a 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;

```