HLA Pattern Matching Facilities — Chapter Five

5.1 Chapter Overview

The HLA Standard Library includes a module that implements a pattern matching domain specific language that supports backtracking. Although the other pattern matching techniques appearing in this chapter are often more efficient that the pattern matching routines in the Standard Library, the HLA Standard Library pattern matching routines are very easy to use; in fact, you have to move up to a very high level language (VHLL) like SNOBOL or Icon in order to find as expressive a pattern matching system. This chapter discussese this powerful language within HLA.

5.2 Introduction to Pattern Matching in HLA

To fully appreciate the power of HLA's pattern matching language you must first learn some background information. This section will provide the background theory necessary to understand how to use each of the pattern matching functions that each of the following sections present.

The basic pattern matching operation is a high-level control structure that the HLA Standard Library patterns module provides. This control structure takes the following form:

```
pat.match( stringValue );  // stringValue is either a string constant or
                           // an HLA string variable

    << sequence of pattern matching functions >>

    << Code to execute if the above sequence succeeds in matching the string >>

  pat.if_failure

    << Code to execute if the above sequence fails to match the string >>

pat.endmatch;
```

The important thing to note from this example is that the pattern matching code is not a function that returns true or false to denote success or failure. Instead, if the set of pattern matching operations succeeds, control falls through to some code to execute on success. On the other hand, if the pattern matching operations fail to match the string, then control falls through to the statements following the *pat.if_failure* clause. Note that if the code successfully matches the string, then the program automatically jumps over the failure section after executing the code associated with the successful pattern match. This behavior is very similar to the program jumping over the ELSE clause of an IF statement.

To give a more concrete example, consider the *pat.matchStr* pattern matching function. This function requires the following syntax:

```
pat.matchStr( StringtoMatch );
```

The *StringToMatch* parameter can be any HLA string variable, constant, or other string object that is legal in a parameter list. If *pat.matchStr* successfully matches this string, control flows to the next statement following the call to *pat.matchStr*; if *pat.matchStr* is not successful, control immediately transfers to the *pat.if_failure* clause of the *pat.match..pat.endmatch* statement. A complete example demonstrates how this works:

```
mov( stdin.a_gets(), stringVar );
pat.match( stringVar );

    pat.matchStr( "Hello" );
    stdout.put( "The string began with ’Hello’", nl );

  pat.if_failure

    stdout.put( "The string did not begin with ’Hello’", nl );

pat.endmatch;
strfree( stringVar );
```

This sequence reads a string from the user and then checks to see if it begins with the string "Hello" (note that the string does not have to be equal to "Hello", it only needs to begin with this string).

When the program encounters the *pat.match* statement in the code above, the program initializes some internal state variables and falls through to the *pat.matchStr* pattern matching function. This call checks to see if *pat.match’s* parameter (*stringVar*) begins with "Hello". If this is the case, then the program falls through to the next statement after the call to *pat.matchStr.* However, if *stringVar* does not begin with "Hello" then control transfers to the *pat.if_failure* clause and continues execution there.

The HLA pattern matching statements work quite a bit differently than the HLA Standard Library string functions. Perhaps one of the biggest differences is that the string functions manipulate strings on a more or less independent basis. That is, if you execute the following two calls, one of them will surely return false:

```
        str.eq( stringVar, "Hello" );
        str.eq( stringVar, " There" );
```

However, if you call two pattern matching functions within the *pat.match..pat.endmatch* statement, the second pattern matching call continues where the first leaves off. As an example, consider the following extension of the previous pattern matching code:

```
mov( stdin.a_gets(), stringVar );
pat.match( stringVar );

    pat.matchStr( "Hello" );
    pat.matchStr( " There" );

    stdout.put( "The string began with ’Hello There’", nl );

  pat.if_failure

    stdout.put( "The string did not begin with ’Hello There’", nl );

pat.endmatch;
strfree( stringVar );
```

In this example, if the first call to *pat.matchStr* succeeds, then the second call attempts to match the string " There" picking up where the previous *pat.matchStr* call left off. Therefore, if the user type "Hello There" then both *pat.matchStr* calls will succeed and control falls through to the first *stdout.put* statement above. On the other hand, if either call to *pat.matchStr* fails, then the code above transfers control to the *pat.if_failure* clause. Of course, we could have achieved this same effect using a single *pat.matchStr* call, but this example demonstrates how the pattern matching functions work together.

Whenever you execute the *pat.match* statement, the pattern matching system saves certain state information about the current pattern match. One of the more important values the pattern matching system maintains is the current *cursor position*. A pattern matching cursor is a pointer into the string that the routines are matching. When you first execute *pat.match*, the system initializes this cursor so that it points at the first character in the string. As you execute the pattern matching functions (like *pat.matchStr*) these functions match their operand against the characters at the current cursor position. If the match is successful, the pattern matching system advances the cursor to the character position just beyond the last character the function matches; if the match fails, then the pattern matching function does not change the cursor position (that is, it will still point at the same character position as it did upon entry into the matching function).

When you stick two pattern matching functions back to back, as in the previous example, the first pattern matching function attempts to match the sequence of characters starting with the beginning of the string. If it is successful, it moves the cursor position beyond the characters it matched and the second pattern matching function begins matching at that position. Likewise, if this second call is successful, then the pattern matching system updates the cursor position yet again and control falls through to the next pattern matching routine (if any).

The pattern matching functions maintain the current cursor position in the ESI register. Therefore, the ESI register will point at the next character to check after you successfully return from a pattern matching function. Likewise, upon entry into a pattern matching function, the pattern matching system saves the original cursor position in the EBX register. So on successful return from the pattern matching function EBX will point at the first character of the sequence that the function matches and ESI will point at one character position beyond the string the function matches. In the previous example assume that user enters "Hello There" during execution; between the calls to *pat.matchStr* the EBX register will point at the "H" character and the ESI register will point at the space appearing between "Hello" and "There" (since the first call to *pat.MatchStr* advances the cursor beyond "Hello"). The pattern matching statement's mainenance of ESI ande EBX is quite useful for extracting substrings that match a particular pattern. In fact, the pattern matching system provides a *pat.extract* function specifically for this purpose; but more on that later.

Note that the pattern matching functions only set up ESI and EBX in this manner on a successful match. If failure occurs, you cannot count on the value in EBX being correct and, therefore, you cannot use extraction techniques to extract a substring that did not match some pattern (which would be logically impossible, anyway).

Consider the following sequence in a pattern matching statement:

```
pat.match( SomeString );

    << pattern matching function #1 >>
    << pattern matching function #2 >>
    << pattern matching function #3 >>
                    .
                    .
                    .
    << pattern matching function #n >>

    << success: Code to execute upon successful match >>

  pat.if_failure

    << failure: Code to execute if unsuccessful match >>

pat.endmatch;
```

In order to execute the code with the *success* label, all of the the pattern matching functions must succeed, in sequence. If any one of them fails (and cannot be coerced into succeeding via backtracking), then the entire sequence fails. That is to say, an *and* relationship exists between each of these matching functions. In order for a matching sequence to succeed, the first call must match *and* the second call must match *and* the third call must match, etc. If any one call fails, the entire sequence fails.

The pattern matching functions fully support "goal-oriented backtracking." This means that if the pattern sequence you specify is ambiguous (meaning there is more than one way to match a given pattern) then the system will attempt to successfully match a pattern using backtracking if possible. Technically, therefore, the pattern matching system is nondeterministic because it will successfully match a pattern if some way of matching the string exists (rather than a single, specific, path through the matching algorithm).

In the examples using *pat.matchStr*, there was no opportunity for backtracking because matching a string (or even a sequence of strings using multiple calls to *pat.matchStr*) is always unambiguous. To see how backtracking works, you'll need to learn another pattern matching function; for the purposes of example here, we'll take a look at the *pat.zeroOrMoreCset* function. As its name suggests, this function matches zero or more characters from a character set you specify as a parameter. A typical call to *pat.zeroOrMoreCset* takes the following form:

```
                              pat.zeroOrMoreCset( someCset );
```

This function always succeeds (because it matches zero or more characters, including zero characters). If the cursor points at a sequence of characters in the *someCset* parameter, this function advances the cursor beyond these characters in the string.

Now consider the following pattern matching code:

```
pat.match( SomeString );

    pat.zeroOrMoreCset( {'A'..'Z'} );
    pat.matchStr( "Hello" );
    stdout.put( "Success" nl );

  pat.if_failure

    stdout.put( "failure" nl );

pat.endmatch;
```

This code begins by skipping over any leading upper case alphabetic characters. Then it attempts to match the string "Hello" immediately thereafter. The astute reader should detect a problem here: if *pat.zeroOrMoreCset* matches all the leading uppercase alphabetic characters, then the *pat.matchStr* call must fail since it begins with an uppercase alphabetic character, that *pat.zeroOrMoreCset* has already skipped. In actuality, the *pat.matchStr* function does fail – the first time the system calls it. The pattern matching system's "goal directed evaluation" will step in and invoke backtracking in order to successfully match the string. Therefore, the sequence above will report success.

An example will clarify how the backtracking logic works. Suppose *SomeString* in the previous example contained "ABCHello There" upon execution of the *pat.match* statement. The *pat.zeroOrMoreCset* function will match all the leading uppercase characters at the beginning of the string. This will leave the cursor pointing at the "e" character immediately following "ABCH". When *pat.matchStr* attempts to match "Hello" the operation fails. However, rather than immediately transferring control to the *pat.if_failure* clause, the code will backtrack and request that *pat.zeroOrMoreCset* try a shorter string. The *pat.zeroOrMoreCset* function complies by matching one less character (that is, it matches the string "ABC" rather than "ABCH"). Note that pat.zeroOrMoreCset is still happy because it's matching three upper case alphabetic characters (which is still zero or more characters from the set). On this call, *pat.zeroOrMoreCset* returns with the cursor pointing at the "H" in "Hello" and control falls through to *pat.matchStr* a second time. This time, of course, *pat.matchStr* succeeds and advances the cursor to the space between the "Hello" and "There" substrings.

While all HLA pattern matching library functions support backtracking, only certain functions will match a different string when asked to backtrack. Specifically, backtracking can only yield a successful match if a particular call to a pattern matching function can match strings of different lengths. For example, although you can backtrack across a *pat.matchStr* function call, *pat.matchStr* always matches the same string. So if some function calls depend upon pat.matchStr to backtrack in order to succeed, they will fail unless some routine before pat.matchStr can successfully backtrack. For example, consider the following code:

```
pat.match( "Hello there Hello" );

    pat.zeroOrMoreCset( {'A'..'Z', 'a'..'z', ' '} );
    pat.matchStr( "Hello" );
    pat.matchStr( " there" );
    stdout.put( "Success" nl );

  pat.if_failure

    stdout.put( "failure" nl );

pat.endmatch;
```

The first call to *pat.zeroOrMoreCset* skips all the characters in the string. Therefore, the following call to *pat.matchStr("Hello")* will fail until *pat.zeroOrMoreCset* backtracks to the beginning of the second "Hello" in the string. At that point, the first and second pattern matching calls will succeed. However, the third call (*pat.matchStr(" there")*) fails because the cursor is currently at the end of the string. Ultimately, however, this pattern will succeed because *pat.zeroOrMoreCset* ultimately backs up to the beginning of the string (matching zero characters) and then *pat.matchStr("Hello")* can match the first five characters allowing *pat.matchStr(" there")* to match the next six characters.

As noted above, only certain functions can change the string they match during backtracking. For example, pat.matchStr always matches the same string, so if pattern matching functions following pat.matchStr depend upon that function to change the cursor position in order to suceed, they will fail. The following code demonstrates this problem:

```
pat.match( "Hello Hello there" );

    pat.matchStr( "Hello" );
    pat.matchStr( " Hello" );
    pat.matchStr( " Hello" );
    stdout.put( "Success" nl );

  pat.if_failure

    stdout.put( "failure" nl );

pat.endmatch;
```

The first call to *pat.matchStr* above successfully matches "Hello" and the second call successfully matches " Hello". The third call to *pat.matchStr* fails because " there" is not equal to " Hello". Although the system will attempt to backtrack, unfortunately there is no way for the previous two calls to *pat.matchStr* to match anything differently than they already have, so the system fails to match the string and transfers control to the *pat.if_failure* clause.

5.3 Character Sequences Versus Strings

Thus far you've seen one form of the *pat.match* statement that allows a single operand which must be a pointer to a string object (i.e., an HLA string variable). In actuality, HLA's pattern matching facilities work with sequences of characters, not strings. While a string is a special case of a character sequence, any sequence of characters in memory is a candidate for use by the pattern matching routines. The pat.match clause provides a special version that allows you to specify the sequence by providing a string operand. This is convenient because most character sequences you want to test will be found in an HLA string. However, there are lots of times when the data you want to test is either not in a string variable or is only a small portion of some larger string. Fortunately, the *pat.match* statement provides additional calling syntax that lets you specify the character sequence by providing two pointers: one to the beginning of the sequence and one to the end of the sequence.

The pattern matching functions (e.g., *pat.matchStr*) always assume that ESI (the cursor position) points at the start of a sequence and EDI points just beyond the end of the sequence. These functions do not assume that ESI points at an HLA string; in particular, the four bytes immediately prior to where ESI points is not the length value. This is obvious if you think about it for a moment. After all, if each matching function increments the cursor pointer (ESI) as it matches characters, then ESI winds up pointing at characters in the middle of the string as the pattern matching operation progresses.

The pattern matching functions use the value in EDI to determine when they've reached the end of the string. While there are two other ways to determine the end of an HLA string (the string's length value or by encountering a zero terminating byte), using a pointer to the first position beyond the end of the sequence to mark the end is a bit more general. This allows you, for example, to match against a short sequence of characters in the middle of a larger string by simply pointing ESI at the first byte of the sequence and EDI at the first character beyond the end of the sequence. To allow the use of a general chracter sequence, the *pat.match* statement provides the following syntax:

```
pat.match( StartOfSequence, EndOfSequence );

   <<Matching functions>>

   << Code to execute if successful match >>

  pat.if_failure;

   << Code to execute if unsuccessful match >>

pat.endmatch;
```

The *StartOfSequence* and *EndOfSequence* operands must be double word pointers to the first character and just beyond the last character of the sequence of characters to match against. *StartOfSequence's* value becomes the initial cursor position. If, during matching, the cursor position becomes greater than or equal to *EndOfSequence*, then the matching fails (or invokes backtracking in an attempt to succeed). Note that if you specify a single (string) operand, *pat.match* initializes the cursor with the address of the first character and adds the length of the string to this address to get the *EndOfSequence* value.

5.4 System Resources During a Pattern Matching Operation

As you've seen already, the pattern matching statements and functions make use of the EBX, ESI, and EDI registers. There are some other serious considerations of which you must be aware when using the HLA pattern matching statements. This section will describe some of those issues.

Since the pattern matching functions pass values between themselves in the EBX, ESI, and EDI registers, you should not modify the values of these registers between calls to the pattern matching functions within a *pat.match..pat.endmatch* statement. While it is perfectly legal to place assembly language statements between pattern matching function calls, you must take care when doing so; otherwise you may inadvertently affect the pattern matching operation in progress.

The pattern matching statement and the pattern matching functions manipulate the stack in very non-standard ways. There are two things you must keep in mind when using the pattern matching functions: first, the *pat.match* statement pushes data on the stack that the *pat.endmatch* clause removes. Therefore, you cannot push data on the stack before a *pat.match* statement and expect that data to be on the top of stack within the *pat.match..pat.endmatch* statement. That is, the following will not produce desired results (and, in fact, it will probably crash the system):

```
// WARNING! This code will not work!

    push( eax );   // Put some value on TOS.
    pat.match( SomeString );

        pop( eax );  // Retrieve value pushed earlier.
         .
         .
         .
    pat.endmatch;
```

The second thing you need to realize is that the pattern matching function calls within the *pat.match..pat.endmatch* statement do not clean up the stack when they return. They must leave certain data on the stack in order to implement backtracking. The pattern matching system cleans up this excess stack data when it encounters the *pat.endmatch* clause; but inside the *pat.match..pat.endmatch* statement, the system takes considerable liberty with how it uses the stack. Therefore, you cannot write code like the following:

```
pat.match( SomeString );
```

```
        pat.zeroOrMoreCset( {'a'..'z'} );
        push( esi );  // Save cursor position  [WARNING! This will NOT work!]
        pat.oneOrMoreCset( {'A'..'Z'} );
        pop( esi );   // Retrieve former cursor position.
         .
         .
         .
    pat.endmatch;
```

The sequence above will not work because ESP is not pointing at the same location after the call to *pat.oneOrMoreCset*[1] as it was before the call. The *pat.oneOrMoreCset* function leaves extra information sitting on the stack in order to support backtracking. Therefore, this code does not pop the original ESI value off the stack, instead, it pops some of the extra information that *pat.oneOrMoreCset*. This may cause the pattern matching operation to fail and may even crash the system. So don't do it!

Of course, another obvious resource that the pattern matching routines use is stack space. While most pattern matches will not consume an inordinate amount of stack space, do be aware that a successful pattern matching operation will require a small amount of stack space[2].

5.5 Eager Versus Lazy Evaluation

Some pattern matching functions, like *pat.oneOrMoreCset*, can match strings of an arbitrary length (as opposed to functions like *pat.matchStr* that, with a given parameter, always match a string with a specific length). Assuming a character seqeuence begins with *n* characters, each of which is a member of character set you pass as *pat.oneOrMoreCset's* parameter, it is perfectly reasonable to expect this pattern to match those *n* characters and advance the cursor beyond them before the next pattern matching function processes characters in the sequence. Indeed, this is exactly the way *pat.oneOrMoreCset* behaves.

Unfortunately, this behavior isn't always what we want. For example, consider the following pattern sequence:

```
        pat.oneOrMoreCset( {'a'..'z'} );
        pat.matchStr( "hello" );
```

Assuming this pattern is attempting to match a string like "abchello" it should succeed. The *pat.oneOrMoreCset* function should match "abc" (which is certainly one or more lower case characters) and then the *pat.matchStr* function matches the remainder of the string ("hello"). Unfortunately, if *pat.oneOrMoreCset* behaves in an eager fashion and attempts to match all the lower case characters in the string, include the lower case characters that make up the string "hello". After *pat.oneOrMoreCset* matches all the characters in the string, the call to *pat.matchStr* will fail since the empty string is not equal to "hello".

Fortunately, the HLA pattern matching functions don't give up at this point and report failure. The HLA pattern matching system uses backtracking and goal directed evaluation to see if some other division of characters between the two functions might succeed. The HLA pattern matching functions contain special code that let them reenter the previous pattern matching function and tell it, if possible, to back up a little bit and try again[3]. The *pat.oneOrMoreCset* function, for example, backs up one character each time the following pattern matching function fails. The following table shows how the sequence above ultimately matches the string "abchello".

1. This function, by the way, succeeds if it matches one or more characters from the specfied character set.
2. If the pattern matching operation is recursive, then it could require considerable stack space. This depends, of course, on the complexity of the pattern and the size of the string you're matching.
3. The extra data the pattern matching functions leave on the stack provides this capability.

Table 1: Matching the String "abchello"

Character Sequence	pat.oneOrMoreCset Matches	pat.matchStr Matches	Result
abchello	abchello		failure
abchello	abchell	o	failure
abchello	abchel	lo	failure
abchello	abche	llo	failure
abchello	abch	ello	failure
abchello	abc	hello	success

Notice how, with backtracking, it took six attempts in order to match the string. In complex patterns where backtracking occurs frequently, the pattern matching algorithm can be very inefficient, especially if earlier patterns consume a large portion of the character sequence under consideration and you have to backtrack through the entire string[4].

Suppose we modified the *pat.oneOrMoreCset* pattern matching algorithm so that it matched as few characters as possible at each step. In other words, it would match one character on the original call, match two characters on the first backtracking operation, match three characters on the third backtracking operation, etc. If this function operated in this manner, here's how the two pattern matching functions above would match "abchello".

Table 2: Matching the String "abchello"

Character Sequence	pat.oneOrMoreCset Matches	pat.matchStr Matches	Result
abchello	a	bchello	failure
abchello	ab	chello	failure
abchello	abc	hello	success

As you can see, this modification arrived at the success state in half the number of steps.

The first version of *pat.oneOrMoreCset* uses an eager algorithm. That is, it eagerly attempts to match as many characters as possible before it stops. This second version of *pat.oneOrMoreCset* uses a lazy algorithm. That is, it matches as few characters as possible, deferring the matching to whatever matching function follows. If the following pattern matching functions fail, then the lazy algorithm matches one additional character each time backtracking occurs.

In this particular example, lazy evaluation was more efficient than eager evaluation. However, this is not always the case. Had there been 10 lower case alphabetic characters before the "hello" substring, then eager evaluation would have been more efficient. Since the relative efficiencies of eager versus lazy evaluation is data dependent, you'll probably want to choose one algorithm or the other based on your knowledge of the data you intend to process. If you don't know, eager evaluation is probably the best choice since it's probably a litte more intuitive.

4. In fact, in some (very) degenerate cases it is possible for the algorithm to consume 2^n units of time if the string is n characters long. Fortunately, this degenerate case almost never occurs in practice.

The *pat.oneOrMoreCset* procedure implements eager evaluation. A second routine, *pat.l_oneOrMoreCset* implements the same pattern matching operation using lazy evaluation. In general, pattern matching function names of the form *pat.XXXXX* (without an "l_" prefix on the XXXXX portion of the name) implement eager evaluation while those with names like *pat.l_XXXXX* use lazy evaluation. Not all pattern matching functions match different strings in the presence of backtracking, therefore some functions do not have a lazy evaluation form.

5.6 Alternation

Sometimes you might want to specify several alternatives within a pattern. While it is technically possible to put alternative matches in the pat.if_failure section of a pattern matching statement, there is an easier mechanism: use the *pat.alternate* clause in the *pat.match..pat.endmatch* statement. The pat.alternate section lets you try to match an alternative pattern before the pattern matching operation reports failure.

The *pat.alternate* clause is an optional section in the *pat.match* statement (unlike the *pat.if_failure* clause, which must be present). You may have zero or more *pat.alternate* clauses in a pattern matching statement. If any *pat.alternate* clauses appear in the statement, they must all precede the *pat.if_failure* clause. The following code shows the basic layout of a *pat.match..pat.endmatch* statement with one or more *pat.alternate* sections:

```
pat.match( SomeString );

        << Pattern matching functions >>

        << Code to execute if the above patterns match >>

    pat.alternate;

        << Pattern matching functions >>

        << Code to execute if the pattern matching functions above succeed >>

    // Additional, optional, pat.alternate sections using the same syntax
    // as the pat.alternate section above.

    pat.if_failure;

        << Code to execute if none of the above patterns matches the string >>

pat.endmatch;
```

A sequence of pattern matching functions within a single section only succeed if all the matches succeed. As noted earlier, this is equivalent to a logical AND operation. Alternation provides the logical OR capability. With alternation, your pattern matching statement can succeed if any one of several different patterns matches.

A little bit later, when you learn how to write your own pattern matching functions, you'll see how to use alternation to create some very sophisticated patterns. In the meantime, however, it's time to take a look at some of the functions that the HLA pattern matching library already provides for you.

5.7 The HLA Standard Library Pattern Matching Routines

The following sections describe each of the HLA pattern matching functions in detail.

5.7.1 'Cursor' Functions

```
procedure pat.EOS;
procedure pat.atPos( pos:dword );
procedure pat.position( pos:dword );
procedure pat.skip( pos:dword );
procedure pat.getPos( var pos:dword );
```

Note: these functions are not standard procedures. You cannot call them outside the context of a *pat.match..pat.endmatch* statement. Any attempt to do so may crash the system.

The pattern matching cursor functions don't actually match any characters in the source string. Instead, these functions succeed or fail based upon the current position of the cursor within the string.

The *pat.EOS* (end of string) pattern matching function succeeds if the cursor is currently at the end of the string (that is, if ESI is equal to EDI). It fails if this is not the case. As you've probably noticed by now, the pattern matching functions don't consider any characters beyond the last characters in the string that they match. You can use the *pat.EOS* function to verify that there are no more characters left in the input string. For example, the following code checks to see if a string contains some number of spaces or tab characters followed by "Hello" at the end of the string:

```
pat.match( SomeString );

        pat.zeroOrMoreCset( { stdio.tab, ' '} );
        pat.matchStr( "Hello" );
        pat.EOS();

        stdout.put( "Success!" nl );

    pat.if_failure;

        stdout.put( "Failure" nl );

pat.endmatch;
```

Note that the pat.EOS pattern does not match a zero terminating byte. It simply checks to see if ESI is (greater than or) equal to EDI (which means the cursor is beyond the end of the string). Remember, the pattern matching functions don't make any assumptions about the string format. To the pattern matching routines, a string is just a sequence of bytes between the locations where ESI and EDI point.

The *pat.atPos* function requires a single *uns32* parameter. This function succeeds if the cursor is currently at the specified character position beyond the start of the original string. For example, *pat.atPos(5)* succeeds if and only if the cursor is currently pointing at the sixth character in the string (i.e., the character who is five bytes beyond the start of the string). This function is quite useful for verifying that certain data (that you match immediately following the *pat.atPos* call) is at a fixed position within a string. For example, the following code matches any string that contains alphabetic characters with the substring "the" starting at offset two in the string:

```
pat.match( SomeString );

    pat.zeroOrMoreCset( {'a'..'z', 'A'..'Z' } );
    pat.atPos( 2 );
    pat.matchStr( "the" );
        .
        .
        .
pat.endmatch;
```

The *pat.position* function function also requires a single *uns32* parameter. If the current sequence of characters is at least this many characters long (i.e., EDI-ESI is greater or equal to the value of this parameter) then this function succeeds and moves the cursor to the specified location in the character sequence. This function fails if the new position is outside the range of the character sequence. Note that this function allows you to move the cursor anywhere in the sequence, backwards or forwards. Example:

```
pat.match( SomeString );

    pat.matchStr( "Name:" );
    pat.position( 4 );
    pat.matchStr( ":John" );
     .
     .
     .
pat.endmatch;
```

The example above matches strings that begin with "Name:John".

The *pat.skip* function has a single *uns32* parameter. This function advances the cursor the specified number of characters this parameter specifies and succeeds if the cursor position is still within the character sequence the system is processing; this function fails if advancing the cursor would move it beyond the end of the string. Note that you can only move forward with this function, you cannot skip backwards with *pat.skip*. Example:

```
pat.match( SomeString );

    pat.matchStr( "Hello" );
    pat.skip( 7 );
    pat.matchStr( "John " );

    stdout.put( "Matched ’HelloxxxxxJohn’" nl );

  pat.if_failure;

    stdout.put( "Failed to match the string" nl );

pat.endmatch;
```

The *pat.getPos* function has a single pass by reference uns32 parameter. This function always succeeds and it stores the offset of the current cursor position into the reference parameter. This matching function is quite useful for noting the position of various sub-patterns in a string. Example:

```
pat.match( SomeString );

    pat.oneOrMoreCset( {’a’..’z’, ’A’..’Z’, ’ ’} );
    pat.getPos( i );
    pat.matchStr( "hello" );

    stdout.put( "Found ’hello’ at position ", i, " in the string" nl );

  pat.if_failure;

    stdout.put( "Did not match the string" nl );

pat.endmatch;
```

5.7.2 Backtracking Control

```
procedure pat.fail;
procedure pat.fence;
```

Note: these functions are not standard procedures. You cannot call them outside the context of a *pat.match..pat.endmatch* statement. Any attempt to do so may crash the system.

The *pat.fail* and *pat.fence* functions provide a small amount of control over backtracking. The *pat.fail* function always fails whereas the pat.fence function always succeeds in one direction (normal execution) and always fails (completely) if you attempt to backtrack across it.

The *pat.fail* function is useful if you have a prefix string you would like to match but want to reject that string if it has a certain suffix. For example, suppose you want to accept all strings that begin with "Hello" unless that string also has " there" immediately following. You can achieve this with the following pattern:

```
pat.match( SomeString );

    pat.matchStr( "Hello" );
    pat.matchStr( " there" );
    pat.fail();

    // NOTE: we’ll never get to this point since pat.fail always fails.

  pat.alternate;

    pat.matchStr( "Hello" );

    stdout.put
    (
        "Matched a string beginning with ’Hello’ that does not " nl
        "begin with ’Hello there’" nl
    );

  pat.if_failure;

    stdout.put
    (
        "String did not begin with ’Hello’ or it began with ’Hello there’"
        nl
    );

pat.endmatch;
```

The *pat.fence* function always succeeds whenever the program encounters it in a sequence of pattern matching functions. However, if the program attempts to backtrack across a *pat.fence* call, the system immediately fails and stops all further attempts at backtracking. This is useful, for example, if you want to skip over certain patterns before processing that pattern. As an example, consider the following code that succeeds if and only if the string ’there’ follows the last ’Hello’ in the string:

```
pat.match( SomeString );

        // Locate the last "Hello" in the string:

        pat.zeroOrMoreCset( {’a’..’z’, ’A’..’Z’, ’ ’} );
        pat.matchStr( "Hello" );

        // Don’t allow backtracking over the last "Hello" string:

        pat.fence();

        // Match a "there" following the last "Hello"

        pat.zeroOrMoreCset( {’a’..’z’, ’A’..’Z’, ’ ’} );
        pat.matchStr( "there" );
        stdout.put( "The last 'Hello' has a 'there' after it.", nl );

    pat.if_failure

        stdout.put( "The last 'Hello' did not have a 'there' following it." nl );

pat.endmatch;
```

5.7.3 Parenthetical Pattern Matching

```
macro pat.onePat / pat.endOnePat
macro pat.zeroOrOnePat / pat.endZeroOrOnePat
macro pat.zeroOrMorePat / pat.endZeroorMorePat
macro pat.oneOrMorePat / pat.endOneOrMorePat
```

Note: these are multipart macros.

These macros accept a general seqeuence of pattern matching function calls between the beginning macro and the corresponding terminator macro. Beyond the obvious use of the latter three "functions" to simulate selection or repetition of a pattern, these macros let you group a set of patterns together so that they succeed or fail as a whole unit. This is comparable to using parentheses in an arithmetic expression to override precedence.

The *pat.onePat* macro evaluates the pattern matching calls appearing in its parameter list and succeeds if they succeed or fails if the sequence fails. This macro is exactly equivalent to parentheses in a regular expression. Here's a simple example of this macro in use:

```
pat.match( SomeString );

        pat.onePat

            pat.zeroOrMoreCset( {'a'..'z'} );
            pat.matchStr( "Hello" );

        pat.endOnePat;

        stdout.put( "String contains 'Hello'" nl );

    pat.if_failure

        stdout.put( "Failure" nl );

pat.endmatch;
```

Of course, there is absolutely no benefit to using the *pat.pattern* macro in this manner. It turns out, however, that you can place a *pat.alternate* section within this macro invocation. This lets you create patterns like the following:

```
pat.match( SomeString );

        // Match "Black" or "Blue":

        pat.onePat

            pat.matchStr( "Black" );
          pat.alternate
            pat.matchStr( "Blue" );

        pat.endOnePat;
        // Match "bird" or "berry":

        pat.onePat

            pat.matchStr( "bird" );
          pat.alternate
            pat.matchStr( "berry" );

        pat.endOnePat;

        stdout.put( "Matched" nl );

    pat.if_failure

        stdout.put( "Failed to match" nl );

pat.endmatch;
```

This pattern will match "Blackbird", "Bluebird", "Blackberry", or "Blueberry". The code is roughly equivalent to the regular expression "(Black | Blue)(bird | berry)".

Although the example above uses only one optional *pat.alternate* section in the *pat.onePat* invocations, multiple *pat.alternate* sections may appear within the macro invocation. In fact, just about anything you can put in a *pat.match..pat.endmatch* statement, with the exceptions of the *pat.if_failure* and *pat.fence* clauses, may appear in the *pat.onePat* macro invocation. Therefore, we could extend the pattern above by adding another alternate clause to the second *pat.onePat* invocation as follows:

```
pat.match( SomeString );

        // Match "Black" or "Blue":

        pat.onePat

            pat.matchStr( "Black" );

          pat.alternate

            pat.matchStr( "Blue" );

        pat.endOnePat;

        // Match "bird" or "berry":

        pat.onePat

            pat.matchStr( "bird" );

          pat.alternate

            pat.matchStr( "berry" );

          pat.alternate

            pat.matchStr( "sky" );

        pat.endOnePat

        stdout.put( "Matched" nl );

    pat.if_failure

        stdout.put( "Failed to match" nl );
```

```
pat.endmatch;
```

Now the pattern matches Blackbird, Blackberry, Blacksky, Bluebird, Blueberry, and BlueSky.

The *pat.zeroOrOnePat* macro also accepts a general sequence of pattern matching statements as a parameter. Like *pat.onePat*, the *pat.zeroOrOnePat* macro allows optional *pat.alternate* sections. The difference between the two calls (which should be obvious from the name) is that *pat.zeroOrOnePat* always succeeds; that is, matching the pattern that is the *pat.zeroOrOnePat* parameter is optional. If r represents some regular expression (or context-free grammar, for that matter) that the *pat.zeroOrOnePat* parameter matches, then the *pat.zeroOrOnePat* invocation is equivalent to the regular expression:

$$(r \mid \varepsilon)$$

The *pat.zeroOrMore* pattern matching macro provides the obvious extension; it matches zero or more occurrences of the pattern you provide as the macro's parameter. This macro always succeeds (since it can match zero occurrences of the pattern); if it does match some pattern, it will advance the cursor beyond the characters it matches (and succeed). If *r* is a regular expression (or CFG), then *pat.zeroOrMore(r)* is equivalent to the following regular expression:

$$(r)^*$$

Warning: this macro consumes a small amount of stack space (about 20 bytes) for each copy of the pattern it matches. If you have a very long character sequence and you expect the pattern to match several times in succession, be aware that this call may use up a considerable amount of stack space. By default, HLA allocates a large stack (16 megabytes) for your programs, so you probably don't have to worry about running out of stack space; however, be aware of the resource usage of this statement, especially if you override HLA's default stack size.

The *pat.oneOrMorePat* macro makes the obvious extension to the *pat.zeroOrMore* macro invocation. Unlike *pat.zeroOrOnePat* or *pat.zeroOrMorePat*, this macro doesn't always succeed. It must match at least one copy of the pattern its parameter specifies. Once it matches at least one copy, the macro returns success and it will consume all addition characters in the sequence that also match the pattern. Like the previous macros this section discusses, the parameter may pat.alternate and other pattern matching functions with the exception of the *pat.if_failure* clause. If *r* is a regular expression (or CFG) that corresponds to the pattern that *pat.oneOrMorePat* matches, then this macro is equivalent to the following regular expression:

$$(r)^+$$

In general, you should not use the *pat.zeroOrOnePat*, *pat.zeroOrMorePat*, or *pat.oneOrMorePat* macros to extend individual pattern matching functions. For example, you should not make the following call:

```
pat.oneOrMorePat;
    pat.oneChar( 'a' );
pat.endOneOrMorePat;
```

(An you can probably figure out, *pat.oneChar* matches a single character, the character its parameter specifies.)

While the call above will correctly match one or more copies of the pattern (which matches a single 'a', hence this pattern matches one or more 'a' characters), you'll discover that most pattern functions fall into a group of related pattern matching functions and there's usually a *pat.oneOrMoreXXXX* version of that function already. For example, the character pattern matching functions include a *pat.oneOrMoreChar.* Calling the function specifically created for this purpose is quite a bit more efficient than wrapping some other function with *pat.oneOrMorePat* (or some other parenthetical pattern macro).

5.7.4 Character Set Pattern Matching Routines

```
procedure pat.peekCset( cst:cset );
procedure pat.oneCset( cst:cset );
procedure pat.upToCset( cst:cset );
procedure pat.zeroOrOneCset( cst:cset );
procedure pat.l_ZeroOrOneCset( cst:cset );
procedure pat.zeroOrMoreCset( cst:cset );
procedure pat.l_ZeroOrMoreCset( cst:cset );
procedure pat.oneOrMoreCset( cst:cset );
procedure pat.l_OneOrMoreCset( cst:cset );
procedure pat.exactlyNCset( cst:cset; n:uns32 );
procedure pat.firstNCset( cst:cset; n:uns32 );
procedure pat.norLessCset( cst:cset; n:uns32 );
procedure pat.l_NorLessCset( cst:cset; n:uns32 );
procedure pat.norMoreCset( cst:cset; n:uns32 );
procedure pat.l_NorMoreCset( cst:cset; n:uns32 );
procedure pat.ntoMCset( cst:cset; n:uns32; m:uns32 );
procedure pat.l_NtoMCset( cst:cset; n:uns32; m:uns32 );
procedure pat.exactlyNtoMCset( cst:cset; n:uns32; m:uns32 );
procedure pat.l_ExactlyNtoMCset( cst:cset; n:uns32; m:uns32 );
```

Note: these functions are not standard procedures. You cannot call them outside the context of a *pat.match..pat.endmatch* statement. Any attempt to do so may crash the system.

The pattern matching functions this section describes all match patterns based on values appearing in some character set. You've already seen the *pat.oneOrMoreCset* function in examples appearing earlier in this chapter, the remaining functions work in a similar manner. The character set pattern matching functions are among the most versatile and most commonly used pattern matching functions in the HLA Standard Library's pattern module. For that reason, you should study each of the routines in this section very carefully. Another reason for taking a closer look at these routines is because you will see a pattern emerging as well discuss other pattern matching functions in the Standard Library. For example, there is nearly a one-to-one correspondance between the character set matching routines and the character matching routines. Therefore, mastery of these routines all but automatically makes you a master of the character matching routines. This section will cover each of the above routines in depth; later sections in this chapter will refer to this section and say something like "this routine behaves very similar to the *pat.xxxxxCset* function, see the description of that routine for details." Therefore, it makes sense to spend a little more time with this section.

All of the pattern matching routines in this group match a sequence of zero or more characters belong to a character set. Therefore, every one of these functions has a character set operand (*cst*). Additionally, some routines limit the number of characters they process to some range of values. Those routines also require one or two additional numeric operands (*n* and *m* in the declarations above). Each of the following subsections will describe the use of these parameters in detail.

If you scan through the list you will notice that there aren't any routines of the form *pat.notInCset*[5]. That's because these routines are not necessary. For example, if you wanted to match one or more characters that are not in a given character set, you could still use the *pat.oneOrMoreCset*. The only difference is that you would match the complement of the character set (that is, match the characters that are not in the set). The HLA Standard Library CSET module provides a routine to take the complement of a character set variable. Of course, if your parameter is a character set constant, you can complement it with the "-" operator.

5. Technically, the *pat.uptoCset* function skips over characters that are not in a character set, but we'll ignore this exception to the rule for now.

5.7.4.1 pat.peekCset

```
procedure pat.peekCset( cst:cset );
```

This pattern matching function succeeds if the character at the current cursor position is a member of the *cst* character set; it fails otherwise. This function does not advance the cursor position, so the next pattern matching routine will reuse the character that this function matches.

This function is useful for "looking ahead" into the character sequence in order to make some decision about whether to proceed with some match. This is especially useful if you've already matched a pattern thus far in the current character sequence but you want to see if you can extend the match by checking to see what appears at the cursor position.

5.7.4.2 pat.oneCset

```
procedure pat.oneCset( cst:cset );
```

This function succeeds if the character at the current cursor position is a member of the *cst* character set. This function fails otherwise. If *pat.oneCset* succeeds, then the system advances the cursor over the character it matches.

5.7.4.3 pat.upToCset

```
procedure pat.upToCset( cst:cset );
```

This function skips over all characters that are not in the character set until it encounters a character that is in *cst*. If none of the characters between the cursor position and the end of the character sequence are members of *cst*, then this function fails. Note that this function will still succeed if the character at the cursor position in a member of *cst* (in which case this function matches zero characters).

RLH - Check This

Note: this function leaves the cursor pointing at the first character in *cst* that it finds. It does not consume thaat character. Hence, the next pattern matching function in the current pattern will have to match and consume that character. Therefore, this function is roughly equivalent to calling *pat.zeroOrMoreCset* with the complement of *cst*. However, you will use this function often enough that it's nice to have a specific function for this purpose.

5.7.4.4 pat.zeroOrOneCset / pat.l_ZeroOrOneCset

```
procedure pat.zeroOrOneCset( cst:cset );
procedure pat.l_ZeroOrOneCset( cst:cset );
```

These functions optionally match a single character at the cursor position if it is a member of the *cst* character set. The *pat.zeroOrOneCset* function eagerly matches the character; that is, if the character at the current cursor position is a member of *cst*, this function will advance the cursor and move it back only upon failure of the following pattern matching routines. The *pat.l_zeroOrMoreCset* function uses lazy evaluation; it will not advance the cursor until the following routines fail and backtrack back into this routine.

You can use these two routines to (optionally) skip over a character that is not a member of some character set by complementing that set and passing the complement as the character set parameter.

5.7.4.5 pat.zeroOrMoreCset / pat.l_ZeroOrMoreCset

```
procedure pat.zeroOrMoreCset( cst:cset );
procedure pat.l_ZeroOrMoreCset( cst:cset );
```

These two functions match a sequence of zero or more characters belonging to the *cst* character set. They always succeed (even if they match zero characters). These functions leave the cursor pointing at the first character that is not a member of *cst* or at the end of the current sequence

The *pat.zeroOrMoreCset* function matches as many characters as possible on the initial call (eager evaluation) and backs off if the following pattern matching functions fail and backtrack back into the routine. The *pat.l_zeroOrMoreCset* function uses lazy evaluation, matching as few characters as possible (starting with zero) and matching as few characters as possible in *cst* as backtracking occurs.

These functions are equivalent to the Kleene Star in regular expression notation. That is, you can easily encode a regular expression like:

```
[a-z]*
```

using the function call "pat.zeroOrMoreCset({'a'..'z'}):" (or the equivalent call to *pat.l_zeroOrMoreCset*).

5.7.4.6 pat.oneOrMoreCset / pat.l_OneOrMoreCset

```
procedure pat.oneOrMoreCset( cst:cset );
procedure pat.l_OneOrMoreCset( cst:cset );
```

The *pat.oneOrMoreCset* and *pat.l_oneOrMoreCset* functions match one or more characters from the *cst* character set. They succeed if they match at least one character in the specified set; they fail if the cursor is pointing at a character that is not a member of *cst*. These functions leave the cursor pointing at the first character that is not a member of *cst* or at the end of the current sequence.

The *pat.oneOrMoreCset* function matches as many characters as possible on the initial call (eager evaluation) and backs off if the following pattern matching functions fail and backtrack back into the routine. The *pat.l_oneOrMoreCset* function uses lazy evaluation, matching as few characters as possible (starting with one) and matching one additional character in *cst* as backtracking occurs.

A call of the form "pat.oneOrMoreCset({'a'..'z'});" is equivalent to the following regular expression:

$$[a\text{-}z]^{+}$$

5.7.4.7 pat.exactlyNCset

```
procedure pat.exactlyNCset( cst:cset; n:uns32 );
```

This function is a little different than the previous functions because it has two parameters. Like the other character set pattern matching functions, *pat.exactlyNCset* has a cst parameter providing the character set to match against. This function also has a second parameter, *n*, that specifies how many characters to match. As the function name suggests, this function succeeds if there are exactly *n* characters, starting at the current cursor position, that are members of the *cst* character set.

Note: this function fails if it can match more than *n* characters. Therefore, either the $n+1^{th}$ character must not be a member of *cst*, or the current sequence ends after *n* characters. If you want to match the first n characters and you don't care about what follows, use the *pat.firstNCset* function (see the next section).

5.7.4.8 pat.firstNCset

```
procedure pat.firstNCset( cst:cset; n:uns32 );
```

This function succeeds if the current character sequence (beginning with the cursor position) contains at least *n* characters that are in *cst*. It fails if there are fewer than *n* characters belonging to *cst*. This function will only consume the first *n* characters of the sequence if it succeeds. It will leave the cursor pointing at the *n+1*th character even if this character is a member of *cst*. Use the *pat.exactlyNCset* function if you want to fail if there are more than *n* characters that belong to *cst*.

5.7.4.9 pat.norLessCset / pat.l_NorLessCset

```
procedure pat.norLessCset( cst:cset; n:uns32 );
procedure pat.l_NorLessCset( cst:cset; n:uns32 );
```

These functions match up to *n* characters that belong to a character set. They will succeed even if they match zero characters. These functions fail if they could match more than n characters in the *cst* parameter.

The *pat.norLessCset* function uses eager evaluation. It will match as many characters as possible and then back off from this if the following pattern matching functions fail and backtrack into *pat.norLessCset*. The *pat.l_NorLessCset* function matches as few characters as possible (starting with zero) and matches additional characters as backtracking occurs.

5.7.4.10 pat.norMoreCset / pat.l_NorMoreCset

```
procedure pat.norMoreCset( cst:cset; n:uns32 );
procedure pat.l_NorMoreCset( cst:cset; n:uns32 );
```

The *pat.nOrMoreCset* and *pat.l_NorMoreCset* functions match a sequence of characters that contain at least *n* characters from the *cst* character set. These functions move the cursor past all characters that they match. They succeed if there are at least *n* characters in *cst*, they fail if there are fewer than *n* characters from *cst* in the character sequence. The *pat.norMoreCset* function uses eager evaluation and matches as many characters as possible, backing off one character at a time if backtracking occurs. The *pat.l_NorMoreCset* functions uses lazy evaluation; it matches only *n* characters (if possible) and then matches additional characters, one at a time, when backtracking occurs.

5.7.4.11 pat.ntoMCset / pat.l_NtoMCset

```
procedure pat.ntoMCset( cst:cset; n:uns32; m:uns32 );
procedure pat.l_NtoMCset( cst:cset; n:uns32; m:uns32 );
```

The pat.ntoMCset and pat.l_NtoMCset functions have three parameters: a character set (*cst*) and two unsigned integer values (*n* and *m*). These two functions succeed if they match at least *n* character from cst starting at the current cursor position in the character sequence. They fail if they cannot match at least *n* characters. These functions will match a maximum of *m* characters from the current sequence. Additional characters beyond the *m*th position may be from the cst character set, but these functions will only advance the cursor beyond the *m*th character they match. These functions still succeed if they match *m* characters.

The *pat.ntoMCset* function eagerly matches characters in the current sequence. If there are at least *m* characters from *cst* in the current sequence, then this function will match them all and back off one character at a time if backtracking occurs. The *pat.l_NtoMCset* function will only match *n* characters and then match additional characters (up to *m* characters) one at a time when backtracking occurs. Neither function will attempt to match more than *m* characters, even if backtracking occurs. Should this situation arise, these functions will backtrack into the previous function and let it deal with the failure.

5.7.4.12 pat.exactlyNtoMCset / pat.l_ExactlyNtoMCset

```
procedure pat.exactlyNtoMCset( cst:cset; n:uns32; m:uns32 );
procedure pat.l_ExactlyNtoMCset( cst:cset; n:uns32; m:uns32 );
```

These functions match between *n* and *m* characters in cst starting at the current cursor position. They return success if they match between *n* and *m* characters and, assuming they match *m* characters, the *m+1*st character in the sequence is not a member of *cst*. This is the main difference between these two functions and the functions in the previous section.

The *pat.exactlyNtoMCset* function uses eager evaluation. It will match as many characters as possible (up to *m* characters) and then back off one character at a time if the following pattern matching routines fail and require backtracking. If it backtracks below *n* characters, then *pat.exactlyNtoMCset* returns failure to the previous pattern matching function. The *pat.l_exactlyNtoMCset* function uses lazy evaluation. Initially, it only matches *n* characters and then matches additional characters (one at a time) as backtracking occurs. It returns failure to its predecessor if it matches *m* characters and the following pattern matching returns still return failure.

5.7.4.13 Sample Program for Character Set Pattern Matching Functions

Program 12.1 Character Set Pattern Matching Functions

5.7.5 Character Pattern Matching Routines

```
procedure pat.peekChar( c:Char );
procedure pat.oneChar( c:Char );
procedure pat.upToChar( c:Char );
procedure pat.zeroOrOneChar( c:Char );
procedure pat.l_ZeroOrOneChar( c:Char );
procedure pat.zeroOrMoreChar( c:Char );
procedure pat.l_ZeroOrMoreChar( c:Char );
procedure pat.oneOrMoreChar( c:Char );
procedure pat.l_OneOrMoreChar( c:Char );
procedure pat.exactlyNChar( c:Char; n:uns32 );
procedure pat.firstNChar( c:Char; n:uns32 );
procedure pat.norLessChar( c:Char; n:uns32 );
procedure pat.l_NorLessChar( c:Char; n:uns32 );
procedure pat.norMoreChar( c:Char; n:uns32 );
procedure pat.l_NorMoreChar( c:Char; n:uns32 );
procedure pat.ntoMChar( c:Char; n:uns32; m:uns32 );
procedure pat.l_NtoMChar( c:Char; n:uns32; m:uns32 );
procedure pat.exactlyNtoMChar( c:Char; n:uns32; m:uns32 );
procedure pat.l_ExactlyNtoMChar( c:Char; n:uns32; m:uns32 );

procedure pat.peekiChar( c:Char );
procedure pat.oneiChar( c:Char );
procedure pat.upToiChar( c:Char );
procedure pat.zeroOrOneiChar( c:Char );
procedure pat.l_ZeroOrOneiChar( c:Char );
procedure pat.zeroOrMoreiChar( c:Char );
procedure pat.l_ZeroOrMoreiChar( c:Char );
procedure pat.oneOrMoreiChar( c:Char );
procedure pat.l_OneOrMoreiChar( c:Char );
procedure pat.exactlyNiChar( c:Char; n:uns32 );
procedure pat.firstNiChar( c:Char; n:uns32 );
procedure pat.norLessiChar( c:Char; n:uns32 );
procedure pat.l_NorLessiChar( c:Char; n:uns32 );
procedure pat.norMoreiChar( c:Char; n:uns32 );
procedure pat.l_NorMoreiChar( c:Char; n:uns32 );
procedure pat.ntoMiChar( c:Char; n:uns32; m:uns32 );
procedure pat.l_NtoMiChar( c:Char; n:uns32; m:uns32 );
procedure pat.exactlyNtoMiChar( c:Char; n:uns32; m:uns32 );
procedure pat.l_ExactlyNtoMiChar( c:Char; n:uns32; m:uns32 );
```

Note: these functions are not standard procedures. You cannot call them outside the context of a *pat.match..pat.endmatch* statement. Any attempt to do so may crash the system.

These functions are very similar to the character set functions the previous sections describe. The difference is that these functions match only a single character (the *c* parameter) rather than any character from a set. Another difference between these functions and the character set functions is that there are two sets of functions above: one set with names like *pat.XXXXChar* and one set with names like *pat.XXXXiChar.* The *pat.XXXXChar* functions match the character you specify as a parameter. The *pat.XXXXiChar* functions also do this, but if the value of the *c* parameter is an alphabetic character, these functions do a case insensitive comparison[6]; that is, they will match an upper or lower case version of *c's* value (note that *c* may contain an upper or lower case character).

While you will not use these functions anywhere near as often as the character set pattern matching functions, do keep in mind that using these functions is a bit more efficient for matching individual characters than calling one of the character set routines with a set containing a single character.

Since you will use these functions infrequently, and you use them just like the character set pattern matching functions, this text will not explain the purpose of these functions on an individual basis. See the corresponding character set routines or the HLA Standard Library documentation for a more detailed description of these routines.

Program 12.2 Demonstration of the Character Based Pattern Matching Functions

5.7.6 String Pattern Matching Routines

```
procedure pat.matchStr( s:string );
procedure pat.matchToStr( s:string );
procedure pat.upToStr( s:string );
procedure pat.matchWord( s:string );

procedure pat.matchiStr( s:string );
procedure pat.matchToiStr( s:string );
procedure pat.upToiStr( s:string );
procedure pat.matchiWord( s:string );

procedure pat.getWordDelims( var cst:cset );  // General procedures, not
```

6. The character set pattern matching routines do not require a case insensitive version because you canalways add both upper and lower case characters to a charac ter set and match characters from both cases at the same time.

```
procedure pat.setWordDelims( cst:cset );      // pattern matching functions.
```

Note: these functions are not standard procedures. You cannot call them outside the context of a *pat.match..pat.endmatch* statement. Any attempt to do so may crash the system.

The string pattern matching routines match substrings within the character sequence beginning at the cursor position within the current character sequence. These functions match a string or some sequence of characters leading up to a string. They succeed if they find the string that their parameter specifies, they fail if the string doesn't exist in the current character sequence. The principle difference between them is the characters they allow before and after the string.

These functions generally come in two forms: the *pat.XXXXStr* functions and the *pat.XXXXiStr* functions. Like the character functions of the previous section, the difference between these two sets of functions is that the *pat.XXXXStr* functions always do an exact string comparison while the *pat.XXXXiStr* functions do a case insensitive comparison.

Two functions, *pat.getWordDelims* and *pat.setWordDelims* aren't actually pattern matching functions. You may call these functions from anywhere within your program (if you call them as part of a pattern matching sequence, they always return success). These functions let you get and set the character set that the *pat.matchWord* and *pat.matchiWord* functions use to delimit "words" during pattern matching. The *pat.getWordDelims* procedure returns a copy of the HLA pattenrn matching library's internal *WordDelims* variable in the pass by reference parameter. The *pat.setWordDelims* copies the parameter's value to the internal *WordDelims* character set variable. By default, this character set all the characters except A-Z, a-z, 0-9, and the underscore ("_") character (that is, -{'A'..'Z', 'a'..'z', '0'..'9', '_'} in HLA set notation). If you want a different set (such as only the whitespace characters) you will probably want to change the value of the WordDelims set using the *pat.setWordDelims* procedure.

5.7.6.1 The pat.matchStr and pat.matchiStr Functions

```
procedure pat.matchStr( s:string );
procedure pat.matchiStr( s:string );
```

The *pat.matchStr* function, which you've already seen, succeeds if the current character sequence begins with the string the *s* parameter specifies. It fails if the character sequence does not begin with the string value of *s*. This function, if successful, advances the cursor position beyond the characters in *s* that it matches.

The *pat.matchiStr* function works in a similar fashion except it compares the leading characters in the character sequence against *s* using a case insenstive comparison. This function succeeds if the only differences are alphabetic case between the *s* and the prefix characters in the current sequence.

5.7.6.2 The pat.matchToStr and pat.matchToiStr Functions

```
procedure pat.matchToStr( s:string );
procedure pat.matchToiStr( s:string );
```

The *pat.matchToStr* function matches all characters in the sequence up to and including the string the *s* parameter. If the function locates a copy of the string s somewhere in the current character sequence (starting with the current cursor position), then this function succeeds and repositions the cursor just beyond the last character it matches. If the string does not exist anywhere in the character sequence (at or beyond the cursor position), then this function fails. This is effectively equivalent to the following pattern sequence:

```
pat.l_zeroOrMoreCset( -{} );   // Equivalent to pat.l_arb();
pat.matchStr( s );
```

The first function call above matches all characters (using a lazy algorithm), therefore, it locates the first substring matching *s* that exists in the current character sequence. Note that there are not eager and lazy versions of this function (this function actually uses a lazy implementation, despite the lack of an "l_" prefix before the name. If you decide you need an eager version of this function, you can easily synthesize it using the following pattern sequence:

```
pat.zeroOrMoreCset( -{} );   // Equivalent to pat.arb();
pat.matchStr( s );
```

Unlike most of the character set functions, lazy evaluation in this case is probably more intuitive than eager evaluation, hence the lack of different functions for the two evaluation mechanisms. Once again, the omission is not a big deal since it is so easy to synthesize the eager evaluation form.

The *pat.matchToiStr* function works in a similar fashion to *pat.matchToStr* except it does a case insenstive comparison. See the above description for further details.

5.7.6.3 The pat.upToStr and pat.upToiStr Functions

```
procedure pat.upToStr( s:string );
procedure pat.upToiStr( s:string );
```

These functions are very similar to the *pat.matchToStr* and *pat.matchToiStr* functions. They succeed and fail under the same conditions. The difference is where these functions leave the cursor if they successfully match the parameter’s value. These functions, if they succeed, leave the cursor pointing at the beginning of the string they match (rather than advancing the cursor beyond that string in the character sequence). These routines are great for skipping over some superfluous data at the beginning of a sequence without removing the string they match from further pattern matching consideration. Of course, the *pat.matchToiStr* function does the same job at *pat.upToStr* except it uses a case insenstive comparison.

5.7.6.4 Example Using the Pattern Matching String Functions

Program 12.3 Pattern Matching Using the String Matching Functions

5.7.7 Extracting Patterns

```
procedure pat.extract( s:string );
procedure pat.a_extract( var s:string );
```

With some pattern matching functions you always know exactly what characters you’ve matched. For example, calls to *pat.oneChar* or *pat.matchStr*, if successful, always match the same substring in the character sequence. Once you get past these calls in some character matching sequence, you intrinsically know the exact data they’ve matched. For many calls, however, this is not the case. Consider *pat.zeroOrMoreCset* or *pat.upToStr*. These functions can match an arbtrary number of (in general) arbitrary characters. If they succeed, the code following their calls really has no clue what characters they’ve matches (or even how many characters they’ve matched). Often, you will need to know what substring some pattern matching function skips over on a successful match. The pattern matching extract functions serve this purpose.

Whenever a pattern matching function succeeds, it returns the current cursor position in ESI and it returns a pointer to the start of the substring it matched in the EBX register. You could manually construct a string immediately after a pattern match by copying the data between these two pointers; however, that would be a lot of work and, besides, the pattern matching library provides two routines to do this for you: the *pat.extract* and *pat.a_extract* functions.

The *pat.extract* function copies the characters between the positions the EBX and ESI specify to the string you pass as a parameter to this function. This string must have sufficient storage allocated to hold at least (ESI-EBX) characters or the function raises a string overflow exception. The results are undefined if the parameter *s* does not contain the address of a value string object in memory. This function returns success and does not affect any register (in particular, it does not affect the current cursor position).

The *pat.a_extract* function allocates storage sufficient storage on the heap to hold the string it extracts. This function stores the address of this string in the pass by reference parameter *s*. Note that, unlike most "a_XXXX" string functions, this function does not return a pointer to the string in the EAX register.

You must take care when using these routines, especially *pat.a_extract*, when backtracking can occur. If backtracking occurs across these functions, they will happily make new copies of each intermediate string on each backtracking operation. This can slow down the pattern matching operation to a crawl if the string you're extracting is rather long. Worse yet, if your code doesn't free the string that *pat.a_extract* allocates, your program will develop a terrific memory leak; that is, you will allocate lots of storage that you can never free, thus using up system memory resources rapidly.

If you must extract the string data from a pattern matching call that the program might repeatedly call because of backtracking, you should simply save the EBX and ESI registers in a couple of local variables and extract the substring at a later (after completing the pattern match) by reloading EBX and ESI and then calling *pat.extract* or *pat.a_extract*.

Note that pat.extract and pat.a_extract are not pattern matching functions, per se. They will, in general, create a string from the characters between EBX and ESI. So in theory, you could call them at any point you've set up these two registers appropriately. However, it is very unusual to call these function outside of a pattern matching sequence or the successful section of a pattern match.

Program 12.4

5.7.8 Matching Arbitrary Patterns

```
procedure pat.arb;
procedure pat.l_arb;
```

Note: these functions are not standard procedures. You cannot call them outside the context of a *pat.match..pat.endmatch* statement. Any attempt to do so may crash the system.

The *pat.arb* and *pat.l_arb* functions will successfully match any sequence of characters. It that sense, they are equivalent to the following function calls:

```
pat.zeroOrMoreCset( -{} );   // Equivalent to pat.arb();
pat.l_zeroOrMoreCset( -{} ); // Equivalent to pat.l_arb();
```

Although semantically these functions are equivalent (that is, they match the same patterns), they have decidedly different implementations. The *pat.arb* and *pat.l_arb* will be significantly faster than the character set matching routines. Like the other *pat.XXXX* and *pat.l_XXXX* pattern matching functions, the two flavors of these routines let you select eager (*pat.arb*) or lazy (*pat.l_arb*) evaluation.

The *pat.arb* function, on its initial call, matches all the characters left in the sequence. While this operation is very efficient (it basically consists of copying EDI's value to ESI), the use of this function can be very inefficient if the following pattern matching functions need to backtrack in order to succeed. So unless you expect the functions following *pat.arb* to match substrings at the end of the current character sequence, you should avoid using *pat.arb.* This function only fails if the pattern matching functions following the call to *pat.arb* return failure (via backtracking) and this function backtracks to the beginning of the character sequence it is processing.

The *pat.l_arb* function, on first call, immediately returns without matching any characters. If the following pattern matching functions fail and backtrack into *pat.l_arb*, then this function consumes a single character on each backtrack operation. This function will usually pass success onto whatever pattern matching functions follow it. If the functions following the call to *pat.l_arb* report failure (via backtracking) and *pat.l_arb* consumes all the characters in the sequence, then this function transmits this failure to the pattern matching function appearing immediately before the call to *pat.l_arb.*

Program 12.5 Example Code Using the Arbitrary Pattern Matching Routines

5.7.9 White Space Pattern Matching Functions

```
procedure pat.getWhiteSpace( var cst:cset );  // General procedures, not
procedure pat.setWhiteSpace( cst:cset );      // pattern matching functions.

procedure pat.zeroOrMoreWS;
procedure pat.oneOrMoreWS;
procedure pat.WSorEOS;
procedure pat.WSthenEOS;
procedure pat.peekWS;
procedure pat.peekWSorEOS;
```

Note: these functions are not standard procedures. You cannot call them outside the context of a *pat.match..pat.endmatch* statement. Any attempt to do so may crash the system.

The pattern matching functions in this section deal with matching "white space" characters or a combination of white space and the end of the string. Although you can easily synthesize these functions from other functions in the pattern matching repertoire, these patterns come up so frequently that it’s convenient to have special functions for them.

By default, the pattern matching library defines "white space" as all control characters (this includes the tab and newline characters), a space, and the DEL character (ASCII code $7F). If you feel that this set is inappropriate (perhaps you only want space, tab, carriage return, and linefeed) you can easily change the value using the *pat.getWhiteSpace* and *pat.setWhiteSpace* functions. These functions are not actually pattern matching functions; they are general purpose functions that you can call outside of a *pat.match..pat.endmatch* statement. If you do call them within a pattern matching sequence, they always return success.

Hint: one common use of the *pat.getWhteSpace* function is to initialize the *WordDelims* (See “String Pattern Matching Routines” on page 1005.) character set to the white space characters. You can easily do this with the following two statements:

```
pat.getWhiteSpace( someCSetVar );
pat.setWordDelims( someCSetVar );
```

5.8 Pattern Matching Sequences

A pattern matching sequence is a set of pattern matching function calls immediately following a *pat.match* or *pat.alternate* statement. The sequence succeeds, and falls through to HLA code following the sequence if and only if all the patterns in the sequence return success. If any single pattern matching function fails, and backtracking does not rectify the situation, then the entire sequence fails and control transfers to the following *pat.alternate* or *pat.if_failure* clause in the *pat.match..pat.endmatch* statement. Although a pattern matching sequence generally consists of pattern matching function calls, it is possible to inject some other x86 instructions into the sequence if you exercise some care. In this section we will explore the issues surrounding this process.

The most important thing to remember is that pattern matching sequences make use of certain CPU resources and you must keep this resource usage in mind when executing code other than pattern matching functions. If you've been reading this chapter straight through, you've already seen the two most important resources the pattern matching routines consume: the x86 registers and stack space. We'll review each of these issues separately in the following paragraphs.

During a pattern matching sequence the pattern matching code uses the EBX, ESI, and EDI registers to maintain pointers into the current character sequence. ESI contains the current cursor position. EBX generally contains a pointer to the beginning of the substring matched by the last pattern matching function, if it was successful (ESI points to just beyond this substring). EDI points to the byte just beyond the end of the character sequence that the current pattern matching sequence is testing. In general, you must not modify the values of these registers during a pattern matching sequence. For example, do not increment ESI to skip over a character in the sequence; instead, call *pat.skip* to do this. Although incrementing ESI has the same effect in many instances, the pat.skip function properly updates internal state variables so the pattern matching operation can properly backtrack over the character you are skipping. Simply incrementing ESI may invalidate the pattern matching system's internal state. For the same reason, you don't want to adjust the end of sequence pointer (EDI) since the pattern matching sequence makes several copies of this value during the normal pattern matching process; tweaking EDI's value may cause the system to produce inconsistent results.

In addition to EBX, ESI, and EDI, pattern matching sequences also make use of other CPU registers. As noted earlier in this chapter, pattern matching functions do not necessarily clean up the stack between pattern matching function calls. Therefore, the value of the ESP register may not be identical to the value in ESP prior to the call. Although the pattern matching functions do not return a specific value in EAX, ECX, or EDX, many pattern matching functions use these registers as scratchpad locations and do not preserve them. All pattern matching functions must preserve EBP's value across a call so that the caller can still access its local variables when the pattern matching function returns.

A direct result of the pattern matching functions' use of ESI and EDI is that you must exercise caution when using HLA class objects in your pattern matching call. Keep in mind that calling class procedures and methods can wipe out the values in the ESI and EDI registers. Therefore, if you use objects in a pattern matching sequence, you will need to preserve these registers' values across the calls. Of course, this same warning applies to any other procedure you call that disturbs the values in the EBX, ESI, and EDI registers.

As noted earlier, the pattern matching functions may leave extra data sitting on the stack when they return. The pattern matching system uses this extra data to implement backtracking and maintain pattern state information. This has two important consequences. First, and probably trivial, a pattern matching sequence will consume some stack space. Unless you have a complex pattern and a long character sequence, this probably isn't an issue (since, by default, HLA programs reserve 16 MBytes of stack space).

The second issue is one you cannot ignore: since the pattern matching functions don't restore the stack between function calls, you cannot push data on the stack before one call and expect to pop that same value off the stack after the call (since the pattern matching function may push additional data onto the stack and leave it there). Worse, if you do push anything onto the stack and leave it there while you call a pattern matching function, you may very well crash the system. This is because the pattern matching functions all assume that the previous pattern matching function has left some data in a certain format sitting on the top of the stack (including backtracking addresses of the previous routine). If you push data onto the stack and then call a pattern matching function, you've effectively scrambled this data structure on the stack and if backtracking occurs the system will jump to the incorrect backtracking address and the system will probably crash.

Note that it is okay to use the stack for temporary use between two pattern matching calls as long as you restore the stack before making the second call. For example, to preserve the ESI and EDI registers across an object's method invocation, you could use code like the following:

```
    pat.somePatternMatchingFunction( --- );

    push( esi );
    push( edi );
    object.methodName( parameters );
    pop( edi );
    pop( esi );

    pat.someOtherPatternMatchingFunction( --- );
```

Since the pattern matching sequence leaves data sitting on the stack that the pat.endmatch clause cleans up, you need to be especially careful about breaking out of a *pat.match..pat.endmatch* statement. In particular, you should be careful about code like the following:

```
forever
     .
     .
     .
    pat.match( someString );

        << Some pattern matching sequence >>

        breakif( some_condition );
            .
            .
            .
      pat.if_failure;
            .
            .
            .
    pat.endmatch;

endfor;
```

The obvious intent in this code is to leave the surrounding FOREVER..ENDFOR loop if the pattern match succeeds (to the point of the BREAKIF statement) and the specified condition is true. Indeed, control will transfer to the first statement beyond the ENDFOR if this is the case. However, since the program doesn't execute the *pat.endmatch* clause in this situation, the program does not get the chance to clean up the stack and restore ESP to its previous value. In some cases this may not harm the system (other than consuming some extra stack space for a short period); in other cases this is a receipe for disaster. If you must break out of a pat.match..pat.endmatch statement, the best solution is to save the value of ESP prior to executing pat.match and restore this value when you break out of the loop. The following code presents one possible way to do this:

```
var
    StkSave:dword;

        .
        .
        .
mov( esp, StkSave );
forever
     .
     .
     .
    pat.match( someString );

        << Some pattern matching sequence >>

        breakif( some_condition );
            .
            .
            .
      pat.if_failure;
            .
            .
            .
    pat.endmatch;

endfor;
mov( StkSave, esp );
```

5.9 Semantic Actions: Acting on the Success of a Pattern Match

For some applications, just knowing whether some string matches a pattern is sufficient. For most applications, however, the program needs to do something special once it matches some portion of the character sequence against a pattern. Borrowing a term from compiler theory, we'll call these activities we must do after a match a *semantic action*.

In one sense, the term semantic action is a fancy term for any x86 code appearing before, within, or after a sequence of pattern matching function calls within a *pat.match..pat.endmatch* statement. Specifically, a semantic action is any code that is not directly related to matching a particular pattern. Even some of the functions the pattern matching library provides fall into this category. Functions like *pat.getWordDelims*, *pat.setWordDelims*, *pat.getWhiteSpace*, *pat.setWhiteSpace*, *pat.extract*, *pat.a_extract* and others are good examples of functions that don't really match anything but, rather, set up values for use inside and outside the pattern matching system.

One of the primary purposes of matching some pattern is to *parse* (or, in plain English, *figure out the meaning of*) some text. Often, the mere successful match of some pattern is sufficient to devine the meaning from some string. In that case whatever code follows the pattern matching sequence intrinsically knows the meaning of the text and can act accordingly. There is need for only a single semantic action, at the end of the pattern matching sequence and the code associated with that action does whatever the pattern dictates.

Sometimes, however, you might actually need to insert semantic actions within the middle of a pattern matching sequence in order to save some data for later processing. For example, suppose you want to allow the user to enter a string of the following form:

```
        <<Numeric Value>> + <<Numeric Value>> = <<Numeric Value>>
```

I.e., you might want to allow the user to enter something like "2+2=4". You can easily do this with the following pattern matching sequence:

```
    pat.oneOrMoreCset( digits );    // Presumably, digits = {'0'..'9'}
    pat.oneChar( '+' );
    pat.oneOrMoreCset( digits );
    pat.oneChar( '=' );
    pat.oneOrMoreCset( digits );
```

This pattern matching sequence will, indeed, match text like the above. Now, however, suppose that you want to validate the user's input from a mathematical standpoint. That is, you only want to report success if they enter a legal text pattern and the sum of the first two integer values equals the third value they input. One easy way to handle this is to use code like the following:

```
const
    digits:cset := {'0'..'9'};
```

```
var
    firstValue: uns32;
    secondValue: uns32;
    thirdValue: uns32;
    s: string;
        .
        .
        .
    pat.oneOrMoreCset( digits );
    pat.a_extract( s );             // Extracts the digit sequence to s.
    conv.strTou32( s, 0 );          // Convert from string to numeric form.
    mov( eax, firstValue );         // Save numeric result.
    strfree( s );                   // Free the storage pat.a_extract allocates.

    pat.oneChar( '+' );

    pat.oneOrMoreCset( digits );    // Process the second integer, as above.
    pat.a_extract( s );
    conv.strTou32( s, 0 );
    mov( eax, secondValue );
    strfree( s );

    pat.oneChar( '=' );

    pat.oneOrMoreCset( digits );    // Process the third integer, as above.
    pat.a_extract( s );
    conv.strTou32( s, 0 );
    mov( eax, thirdValue );
    strfree( s );

    // Okay, we matched the generic text pattern, now verify that arithmetic
    // was reasonable.

    mov( firstValue, eax );
    add( secondValue, eax );
    if( eax = thirdValue ) then

        stdout.put( "The arithmetic expression is correct" nl );

    else

        stdout.put
        (
            "Sorry, ",
            firstValue,
            " + ",
            secondValue,
            " = ",
            (type uns32 eax),
            ", not ",
            thirdValue,
            nl
        );

    endif
```

To be a little more user friendly, this code should probably allow whitespace before and after each of the digit sequences (just call *pat.zeroOrMoreWS*). The implementation of this feature is left as an exercise at for the end of the chapter.

The values held by the *firstValue*, *secondValue*, and *thirdValue* variables are known as attributes (once again, borrowing a term from compiler theory) of the text they match. Don’t get the idea that each pattern matching call has an attribute associated with it. The code above associates attributes with the results it obtains from three of the five pattern matching functions it calls. So, obviously, we don’t have to associate an attribute like a value with each pattern matching call[7]. Sometimes, however, we might want to associate a single attribute with a sequence of pattern matching calls. For example, suppose we want to modify the example above so that it allows the user to enter *int32* values rather than just unsigned values. To accomplish this we have to allow for an optional sign character in front of the digits. One easy way to do this is as follows:

```
pat.pattern
(
    pat.zeroOrOneChar( ’-’ );
    pat.oneOrMoreCset( digits );
);
pat.a_extract( s );
conv.strToi32( s, 0 );
mov( eax, secondValue );
strfree( s );
```

The parenthetical pattern (*pat.pattern*) collects all the characters of the (possibly) signed integer constant into a single string so that *pat.a_extract* has an easier time producing a single string value. Without the pat.pattern macro, you’d have to do something ugly like the following:

```
mov( esi, startPosn );          // Save (possible) start of integer value.
pat.zeroOrOneChar( ’-’ );       // Match optional ’-’.
pat.oneOrMoreCset( digits );    // Process the second integer, as above.
mov( startPosn, ebx );          // Set EBX to point at ’-’ or 1st digit.
pat.a_extract( s );
conv.strTou32( s, 0 );
mov( eax, theValue );
strfree( s );
```

Remember, *pat.a_extract* extracts the characters between EBX and ESI. Unfortunately, each pattern matching function resets EBX to the start of the string that it matches. Therefore, to have EBX and ESI surround the text that two functions match, we have to use a parenthetical pattern matching function or we have to manually save the pointer to the start of the pattern and reload EBX with this value before calling *pat.a_extract*.

Whichever of the previous two solutions you choose, one thing is clear: now an attribute is associated with the string that two successive functions match. Therefore, you should not associate attributes with a single HLA pattern matching function, but with some specific pattern (i.e., regular expression or context free grammar production).

Although the examples above define the integer equivalent of some numeric string as "the" attribute, a given sequences of characters that some pattern matches can have zero, one, or even more attributes associated with it. As briefly noted in a footnote, the string of characters that a pattern matches is also an attribute of some pattern match (the compiler theory term for this string is *lexeme*). In general, you can assign whatever attributes you want to some sequence of characters you match during a pattern matching operation. The important thing to keep in mind is that you must compute and save those attribute values immediately after matching the lexeme on which you base the attributes. In particular, once you begin processing the remainder of the character sequence with additional pattern matching functions the values in EBX and ESI (that you use to extract the information) will be lost.

There is a very big problems with embedding the code for your semantic actions within a pattern matching sequence: what happens in the presence of backtracking and failure? The first problem happens when backtracking occurs: specifically, the program could execute your semantic action over and over again many times before it successfully matches the text and falls through to the success section. At the very least, this

7. Of course, one intrinsic attribute is the text that a function matches; for example, "pat.oneChar(’+’);" matches the plus sign so ’+’ is an implied attribute of the text that this function matches. We’ll ignore this for the time being.

could have a big performance impact on your program. Consider the following slight modification to the previous examples:

```
    pat.pattern
    (
        pat.zeroOrOneChar( '-' );
        pat.l_oneOrMoreCset( digits );
    );
    pat.a_extract( s );
    conv.strToi32( s, 0 );
    mov( eax, secondValue );
    strfree( s );
```

If you missed the difference, that's easy to understand. There are only two characters different between this example: "l_". That is, this function calls *pat.l_oneOrMoreCset* rather than *pat.oneOrMoreCset*. What a difference, however, this might make in the execution of the program. Suppose, for example, that the user inputs a three-digit numeric string for the second value. The lazy evaluation algorithm will require backtracking across this value three times (each time allocating storage, extracting the string, and recomputing the integer equivalent of the value). While this is not an extremely slow operation, doing it three times when once would have been sufficient is clearly a waste.

Worse still, the backtracking can actually create a defect in the program. Suppose the code above wanted to save the lexeme that the pattern matches and does so by not freeing up the string *s* at the end of the sequence. This creates a whopper of a memory leak since, upon each backtracking operation, the program will allocate a new string and lose track of the previous allocation (i.e., you've got a memory leak).

Even if backtracking cannot occur within some pattern (because none of the pattern matching functions support backtracking), you can still get into trouble if several pattern matching functions early in the sequence succeed, execute some semantic actions that allocate storage, and then some later pattern matching function fails and transfers control to a *pat.alternate* section or a *pat.if_failure* section.

The paragraphs above list two problems that can occur during backtracking or failure (loss of efficiency and a memory leak). However, any activity that consumes undo resources, can only execute once, or cannot be easily undone should not appear in an embedded semantic action in the middle of some pattern matching sequence. Placing such code within the matching sequence will prove to be a disaster.

One solution to this problem is to do as little processing as possible in semantic actions appearing in the middle of a pattern matching sequence. Save the real work for the end of the sequence whenever possible. For example, rather than converting our integer values directly to a number and saving that numeric value within the semantic actions, we could modify the code so that it only saves the pointers to the lexemes that the sub-patterns match. At the end of the pattern matching sequence we can use these pointers to extract the strings and translate them to the appropriate integer values. The following code does this with the original *uns32* conversion:

```
const
    digits:cset := {'0'..'9'};
var
    firstValue: uns32;
    secondValue: uns32;
    thirdValue: uns32;
    s: string;

    ebxVals: dword[2];   // Save pointers to the lexemes in these two arrays.
    esiVals: dword[2];
        .
        .
        .
    pat.oneOrMoreCset( digits );
    mov( ebx, ebxVals[ 0*4 ]);      // Save ptr to start of lexeme[0].
    mov( esi, esiVals[ 0*4 ]);      // Save ptr to end of lexeme[0].

    pat.oneChar( '+' );
pat.oneOrMoreCset( digits );    // Process the second integer, as above.
mov( ebx, ebxVals[ 1*4 ]);      // Save ptr to start of lexeme[1].
mov( esi, esiVals[ 1*4 ]);      // Save ptr to end of lexeme[1].

pat.oneChar( '=' );

pat.oneOrMoreCset( digits );    // After this, we're done matching the
mov( stralloc( 64 ), eax );     // pattern so we can immediately process
pat.extract( s );               // its lexeme without first saving the
conv.strTou32( s, 0 );          // pointers.
mov( eax, thirdValue );

// Now compute the numeric values for the first and second lexemes above:

mov( ebxVals[ 0*4 ], ebx );
mov( esiVals[ 0*4 ], ebx );
pat.extract( s );
conv.strTou32( s, 0 );
mov( eax, firstValue );

mov( ebxVals[ 1*4 ], ebx );
mov( esiVals[ 1*4 ], ebx );
pat.extract( s );
conv.strTou32( s, 0 );
mov( eax, secondValue );



// Okay, we matched the generic text pattern, now verify that arithmetic
// was reasonable.  (Note: EAX contains secondValue at this point.)

add( firstValue, eax );
if( eax = thirdValue ) then

    stdout.put( "The arithmetic expression is correct" nl );

else

    stdout.put
    (
        "Sorry, ",
        firstValue,
        " + ",
        secondValue,
        " = ",
        (type uns32 eax),
        ", not ",
        thirdValue,
        nl
    );

endif
```

This example will run correctly even in the face of backtracking or failure. Further, it doesn't consume an excessive number of extra CPU cycles in either of these cases (the only extra work involved is storing away the values of two pointers rather than allocating storage, extracting a string, and converting that string to integer form). The heavy-duty work doesn't occur until the end, once success is assured.

5.10 Writing Your Own Pattern Matching Routines

Although HLA provides a wide variety of pattern matching functions, from which you can probably synthesize any pattern you desire, there are several reasons why you might want to write your own pattern matching routines. Some common reasons include:

(1) You would like a more efficient pattern matching function than is possible by composing existing pattern matching functions.

(2) You need a particular pattern matching routine to produce a side effect and the standard matching routines do not produce the desired side effect. A common example is a pattern matching routine that returns an attribute value for an item it matches. For example, a routine that matches a string of decimal digits may return the numeric equivalent of that string as an attribute of that pattern.

(3) You need a pattern matching routine that considers other machine states (i.e., variable values) besides the string the pattern is processing.

(4) You need to handle some context-sensitive issues.

(5) You want to understand how the pattern matching algorithm works.

Writing your own pattern matching functions can achieve all these goals and many more.

The first issue you must address when writing your own pattern matching routine is whether or not the routine supports backtracking. Generally, this decision depends upon whether the function matches strings that are always a fixed length or can match strings of differing lengths. For example, the pat.oneCset routine always matches a string of length one whereas the pat.zeroOrMoreCset function can match strings of any length. If a function can only match strings having a fixed length, then the function does not need to support back tracking.

Generally, pattern matching functions that can match strings of varying lengths should support backtracking[8] . Since supporting backtracking is more work and less efficient, you should only support it when necessary.

Once you've decided that you're going to support back tracking in a matching function, the next issue that concerns you is whether the function supports eager evaluation or lazy/deferred evaluation. (Note: when writing general matching routines for library use, it's generally a good idea to supply two functions, one that supports eager evaluation and one that supports lazy/deferred evaluation.) A function that supports eager evaluation tries to match the longest possible string when the program calls the function. If the function succeeds and a later matching functions fails (invoking the backtracking operation), then the matching function backs off the minimum number of characters that will still match. This process continues until the following code succeeds or the function backs off so much that it, too, fails.

A function that supports lazy/deferred evaluations tries to match the shortest possible string. Once it matches the shortest string it can, it passes control on to the following pattern matching functions. If they fail and backtracking returns control to the function, it tries to match the next smallest string larger than the one it currently matches. This process repeats until the following match functions succeed or the current function fails to match anything.

Note that the choice of eager vs. lazy/deferred evaluation does not generally affect whether a pattern will match a given string[9] . It does, however, affect the efficiency of the pattern matching operation. Backtracking is a relatively slow operation. If an eager match causes the following pattern functions to fail until the current pattern matching function backs off to the shortest possible string it can match, the program will run much slower than one that uses lazy evaluation for the function (since it starts with the shortest possible string to begin with). On the other hand, if a function needs to match the longest possible string in order for

8. Although this is your decision. If for some reason you don't want to support backtracking in such functions, that is always an option you can choose.

9. The one exception has to do with fences. If you set a fence after the pattern matching routine, then backtracking cannot return into the pattern matching function. In this one case, the choice of deferred vs. eager evaluation will have an impact on whether the whole pattern will match a given string.

the following matching functions to succeed, choosing lazy evaluation is not a wise choice since it will be slower. Therefore, the choice of which form is best to use is completely data dependent. If you have no idea which evaluation form should be better, choose eager evaluation since it is more intuitive to those defining the pattern to match.

All pattern matching routines have two implicit parameters passed to them in the ESI and EDI registers. ESI is the current "cursor" position while EDI points at the byte immediately after the last character available for matching. That is, the characters between locations ESI and EDI-1 form the character sequence to match against the pattern.

The primary purpose of a pattern matching function is to return "success" or "failure" depending upon whether the pattern matches the characters in the string (or however else you define "success" versus "failure"). In addition to returning success or failure, pattern matching functions must also return certain values in some of the registers. In particular, the function must preserve the value in EDI (that is, it must still point at the first byte beyond the end of the string to match). If the function succeeds, it must return EBX pointing at the start of the sequence it matched (i.e., EBX must contain the original value in ESI) and ESI must point at the first character beyond the string matched by the function (so the string matched is between addresses EBX and ESI-1). If the function fails, it must return the original values of ESI and EDI in these two registers. EBX's value is irrelevant if the function fails. Except for EBP, the routine need not preserve any other register values (and, in fact, a pattern matching function can use the other registers to return attribute values to the calling code).

Pattern matching routines that do not support backtracking are the easiest to create and understand. Therefore, it makes sense to begin with a discussion of those types of pattern matching routines. A pattern matching routine that does not support backtracking succeeds by simply returning to its caller (with the registers containing the appropriate values noted above). If the function fails to match the characters between ESI and EDI-1, it must call the pat._fail_ function passing the pat.FailTo object as its parameter, e.g.,

pat._fail_(pat.FailTo);

As a concrete example, consider the following implementation of the pat.matchStr function:

```
unit patterns;

#include( "pat.hhf" );

procedure pat.matchStr( s:string ); nodisplay; noframe;
begin matchStr;

    push( ebp ); // must do this ourselves since noframe
    mov( esp, ebp ); // is specified as an option.
    cld();

    // Move a copy of ESI into EBX since we need to return
    // the starting position in EBX if we succeed.

    mov( esi, ebx );

    // Compute the length of the remaining
    // characters in the sequence we are attempting
    // to match (i.e., EDI-ESI) and compare this against
    // the length of the string passed as a parameter.
    // If the parameter string is longer than the number
    // of characters left to match, then we can immediately
    // fail since there is no way the string is going to
    // to match the string parameter.

    mov( s, edx );
    mov( (type str.strRec [edx]).length, ecx );
    mov( edi, eax );
    sub( esi, eax );
    if( ecx > eax ) then

        // At this point, there aren't enough characters left
        // in the sequence to match s, so fail.

        pat._fail_( pat.FailTo );

    endif;

    // Okay, compare the two strings up to the length of s
    // to see if they match.

    push( edi );
    mov( edx, edi );
    repe.cmpsb();
    pop( edi );
    if( @ne ) then

        // At this point, the strings are unequal, so fail.
        // Note that this code must restore ESI to its
        // original value if it returns failure.

        mov( ebx, esi );
        pat._fail_( pat.FailTo );

    endif;

    // Since this routine doesn't have to handle backtracking,
    // a simple return indicates success.

    pop( ebp );
    ret();

end matchStr;
end patterns;
```

Program 12.6 Implementation of pat.matchStr (non-backtracking function)

If your function needs to support back tracking, the code will be a little more complex. First of all, your function cannot return to its caller by using the RET instruction. To support backtracking, the function must leave its activation record on the stack when it returns. This is necessary so that when backtracking occurs, the function can pick up where it left off. It is up to the *pat.match* macro to clean up the stack after a sequence of pattern matching functions successfully match a string.

If a pattern matching function supports backtracking, it must preserve the values of ESP, ESI, and EDI upon initial entry into the code. It will also need to maintain the currrent cursor position during backtracking and it will need to reserve storage for a special *pat.FailRec* data structure. Therefore, almost every pattern matching routine you'll write that supports backtracking will have the following VAR objects:

```
var
    cursor: misc.pChar;       // Save last matched posn here.
    startPosn: misc.pChar;    // Save start of str here.
    endStr: misc.pChar;       // End of string goes here.
    espSave: dword;           // To clean stk after back trk.
    FailToSave:pat.FailRec;   // Save global FailTo value here.
```

Warning: you must declare these variables in the VAR section; they must not be static objects.

Upon reentry from backtracking, the ESP register will not contain an appropriate value. It is your code's responsibility to clean up the stack when backtracking occurs. The easiest way to do this is to save a copy of ESP upon initial entry into your function (in the *espSave* variable above) and restore ESP from this value whenever backtracking returns control to your function (you'll see how this happens in a moment). Likewise, upon reentry into your function via backtracking, the registers are effectively scrambled. Therefore, you will need to save ESI's value into the *startPosn* variable and EDI's value into the *endStr* variable upon initial entry into the function. The *startPosn* variable contains the value that EBX must have whenever your function returns success.

The *cursor* variable contains ESI's value after you've successfully matched some number of characters. This is the value you reload into ESI whenever backtracking occurs. The *FailToSave* data structure holds important pattern matching information. The pattern matching library automatically fills in this structure when you signal success; you are only responsible for supplying this storage, you do not have to initialize it.

You signal failure in a function that supports backtracking the same way you signaled failure in a routine that does not support backtracking: by invoking "pat._fail_(pat.FailTo);" Since your code is failing, the caller will clean up the stack (including removing the local variables you've just allocated and initialized). If the pattern matching system calls your pattern matching function after backtracking occurs, it will reenter your function at its standard entry point where you will, once again, allocate storage for the local variables above and initialize them as appropriate.

If your function succeeds, it usually signals success by invoking the *pat._success_* macro. This macro invocation takes the following form:

```
pat._success_( FailToSave, FailToHere );
```

The first parameter is the *pat.FailRec* object you declared as a local variable in your function.

The *pat._success_* macro stores away important information into this object before returning control to the caller. The *FailToHere* symbol is a statement label in your function. If backtracking occurs, control transfers to this label in your function (i.e., this is the backtracking reentry point).

The code at the *FailToHere* label must immediately reload ESP from *espSave*, EDI from *endStr*, EBX from *startPosn*, and ESI from *cursor.* Then it does whatever is necessary for the backtrack operation and attempts to succeed or fail again.

The *pat._success_* macro (currently) takes the following form[10]:

```
// The following macro is a utility for
// the pattern matching procedures.
// It saves the current global "FailTo"
// value in the "FailRec" variable specified
// as the first parameter and sets up
// FailTo to properly return control into
// the current procedure at the "FailTarget"
// address. Then it jumps indirectly through
// the procedure's return address to transfer
// control to the next (code sequential)
// pattern matching routine.

macro _success_( _s_FTSave_, _s_FailTarget_ );

    // Preserve the old FailTo object in the local
    // FailTo variable.

    mov( pat.FailTo.ebpSave, _s_FTSave_.ebpSave );
    mov( pat.FailTo.jmpAdrs, _s_FTSave_.jmpAdrs );

    // Save current EBP and failto target address
    // in the global FailTo variable so backtracking
    // will return the the current routine.
```

10. This code was copied out of the "patterns.hhf" file at the time this document was written. You might want to take a look at the patterns.hhf header file to ensure that this code has not changed since this document was written.

```
    mov( ebp, pat.FailTo.ebpSave );
    mov( &_s_FailTarget_, pat.FailTo.jmpAdrs );

    // Push the return address onto the stack (so we
    // can return to the caller) and restore
    // back to the caller without cleaning up
    // the current routine’s stack.

    push( [ebp+4] );
    mov( [ebp], ebp );
    ret();

endmacro;
```

As you can see, this code copies the global *pat.FailTo* object into the *FailToSave* data structure you’ve created. The *FailTo* structure contains the EBP value and the reentry address of the most recent function that supports backtracking. Your code must save these values in the event your code (ultimately) fails and needs to backtrack to some previous pattern matching function. After preserving the old value of the global *pat.FailTo* variable, the code above copies EBP and the address of the *FailToHere* label you’ve specified into the global *pat.FailTo* object.

Finally, the code above returns to the user, without cleaning up the stack, by pushing the return address (so it’s on the top of the stack) and restoring the caller’s EBP value. The RET instruction above returns control to the function’s caller (note that the original return address is still on the stack, the pattern matching routines will never use it).

Should backtracking occur and the program reenters your pattern matching function, it will reenter at the address specified by the second parameter of the *pat._success_* macro (as noted above). You should restore the appropriate register (as noted above) and use the value in the *cursor* variable to determine how to proceed with the backtracking operation. When doing eager evaluation, you will generally need to decrement the value obtained from *cursor* to back off on the length of the string your program has matched (failing if you decrement back to the value in *startPosn*).

When doing lazy evaluation, you generally need to increment the value obtained from the *cursor* variable in order to match a longer string (failing if you increment *cursor* to the point it becomes equal to *endStr*).

When executing code in the reentry section of your procedure, the failure and success operations are a little different. Prior to failing, you must manually restore the value in *pat.FailTo* that *pat._success_* saved into the *FailToSave* local variable. You must also restore ESI with the original starting position of the string. The following instruction sequence will accomplish this:

```
// Need to restore FailTo address because it
// currently points at us. We want to jump
// to the correct location.
mov( startPosn, esi );
mov( FailToSave.ebpSave, pat.FailTo.ebpSave );
mov( FailToSave.jmpAdrs, pat.FailTo.jmpAdrs );
pat._fail_( pat.FailTo );
```

Likewise, succeeding in the backtrack reentry section of your program is a little different. You do not want to invoke the *pat._success_* macro because it will overwrite the *FailToSave* value with the global *pat.FailTo*. The global value, however, points at your routine; were you to overwrite this value you’d never be able to fail back to previous matching functions in the current pattern match. Therefore, you should always execute code like the following when succeeding in the reentry section of your code:

```
mov( esi, cursor );   //Save current cursor value.
push( [ebp+4] );      //Make a copy of the rtn adrs.
mov( [ebp], ebp );    //Restore caller’s EBP value.
ret();                //Return to caller.
```

The following is the code for the *pat.oneOrMoreCset* routine (that does an eager evaluation) that demonstrates pattern matching with backtracking.

```
*** INSERT CODE HERE
```

Program 12.7 Implementation of the pat.oneOrMoreCset Function

The following example code demonstrates the *pat.l_OneOrMoreCset* routine. This is the same routine as the code above except this code supports lazy/deferred evaluation rather than eager evaluation.

Program 12.8 Implementation of the pat.l_OneOrMoreCset Function

5.11 Recursive Pattern Matching Operations

**** Here I Am

5.12 Converting Regular Expressions to HLA Pattern Matching Code

5.13 Converting CFGs to HLA Pattern Matching Code