25 Patterns Module (patterns.hhf)

The HLA Standard Library provides a set of string/pattern matching routines that are similar in use to those provided by the SNOBOL4 and Icon programming languages. These pattern matching routines support recursion and backtracking, allowing the specification of context-free grammars as well as regular expressions.

Note: Because many of the "functions" in the pattern-matching library are actually macro invocations, this document does not provide examples of low-level pattern-matching function calls.

Warning: unlike most HLA Standard Library functions, the pattern matching functions do not preserve all the registers they modify. In fact, EDX is the only register whose value may be preserved; almost all the other registers are used by the pattern matching code and you should expect their values to be modified by the pattern matching functions whenever you call them.

25.1 The Patterns Module

To use the pattern functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "patterns.hhf" )
or
#include( "stdlib.hhf" )
```

25.2 An Introduction to Pattern Matching (a tutorial)

The HLA pattern matching library scans for patterns of characters within a string or within some sequence of characters. A pattern matching operation consists of a sequence of pattern matching commands that execute on the sequence. The result of a pattern matching operation is either *success* (meaning all the pattern matching commands succeeded) or *failure* (meaning at least one of the commands failed to match). The success or failure of a pattern matching operation directs program execution to one of two different locations in the code (not unlike an IF/ELSE/ENDIF statement) so the program can perform different operations based on the success or failure of a pattern match.

You must understand that the HLA Standard Library pattern matching functions don't return true or false that you can test in a conditional expression (e.g., in an IF or WHILE statement). Instead, the pattern matching functions and macros actual introduce a *new control structure* in the HLA language. Within this control structure, the successful execution of each pattern matching operation allows the program to continue execution with the next successive statement in the control structure. However, if the pattern matching operation fails, then control transfers to a different location in the pattern matching control construct. In a sense, this is very similar to HLA's *try..exception..endtry* statement. A failed pattern matching operation transfers control to some distinct failure location (just like an exception occuring in a *try..endtry* block); successful pattern matching operations fall through to the next command or, if all commands in a pattern matching command sequence are successful, control transfers to the first statement after the pattern matching control structure.

The pattern matching control sequence is delimited by the *pat.match* and *pat.endmatch* macro invocations. Between these two statements, exactly one *pat.if_failure* macro invocation must appear. The template of a pattern matching statement is the following:

```
pat.match( <<character sequence to match>> );

  << Sequence of match operations>>

  << Code to execute on a successful match >>

 pat.if_failure

  << Code to execute if the match fails >>

pat.endmatch;
```

The "sequence of match operations" appearing in this control structure is a set of zero or more pattern matching function calls. As noted above, if a given function succeeds, the control falls through to the next command in the control structure (or through to the "code to execute on a successful match" if all of the matching commands succeed). If a match operation fails, the the program immediately transfers control to the code following the *pat.if_failure* statement.

The *pat.match* statement supports two difference syntaxes. The first form accepts a single HLA string object as a parameter. This form is invoked thusly:

```
pat.match( StringValue );
```

Technically, you could supply a string variable or a string constant as this argument. However, it would never make any sense (other than for testing or demonstration purposes) to supply a literal string constant as this argument. The purpose of the pattern matching functions is to determine if some unknown string matches a given pattern. If the string's value is known while you're writing the program, there really isn't any need to do the pattern matching operation – you can do the pattern matching operation in your head and skip the execution of the code. Nevertheless, many of the examples in this document will use literal string constants as the test string in order to make the examples easier to understand.

The second form of the *pat.match* statement expects two arguments. The first is a pointer to the first character of some character sequence you wish to match and the second argument is a pointer to the first byte beyond the end of the character sequence you wish to match. This invocation takes the following form:

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

Note that internally, the *pat.match* macro actually uses the start and end of sequence pointers. If you pass the *pat.match* function a single string argument, *pat.match* uses the string pointer as the *StartOfSequence* pointer and it adds the strings length to the *StartOfSequence* value to obtain the *EndOfSequence* address. For the sake of discussion, we'll call the string (or sequence of characters) we're trying to match the *match sequence*.

During a pattern matching operation, there are three important pointers the functions use: a pointer to the first character of the character sequence, a pointer to the first byte beyond the character sequence, and a *cursor* pointer that points at the next character under consideration. When you invoke *pat.match*, the macro begins by initializing the cursor with the address of the first character in the match sequence (e.g., the *StartOfSequence* value). The pattern matching commands operate on the character data at the current cursor position through the end of the sequence (that is, up to the byte before the address held in the end of sequence pointer). If the cursor's value is ever greater than or equal to the end of sequence value and the program attempts to execute a pattern matching function that would advance the cursor, then the pattern matching operation fails (and control transfers to the *pat.if_failure* statement).

Let's consider a concrete example. The *pat.oneChar* function accepts a single character argument. If the cursor's value is less than the end of sequence value and the character that the cursor points at matches *pat.oneChar's* argument, then the *pat.oneChar* function succeeds and increments the cursor value to skip over the character in the sequence that it matched. The following pattern matching operation succeeds and prints "Encountered 'c'":

```
pat.match( "c" );

  pat.oneChar( 'c' );
  stdout.put( "Encountered 'c'" nl );

  pat.if_failure

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

pat.endmatch;
```

The following code, however, prints "Failed to match 'c'" because the cursor (initialized with the address of the 'd' character) doesn't point at a byte containing 'c':

```
pat.match( "d" );

  pat.oneChar( 'c' );
  stdout.put( "Encountered 'c'" nl );

  pat.if_failure

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

pat.endmatch;
```

Whenever a pattern matching function such as *pat.oneChar* succeeds, it advances the cursor over the character(s) it matches. Upon return from the pattern matching function, any successive calls to a pattern matching function will attempt to match the character(s) immediately after those already matched. For example, consider the following pattern matching construct:

```
pat.match( "cd" );

  pat.oneChar( 'c' );
  pat.oneChar( 'd' );
  stdout.put( "Encountered 'cd'" nl );

  pat.if_failure

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

pat.endmatch;
```

The first call to *pat.oneChar* matches the 'c' in the match sequence and advances the cursor by one position (so that it now points at the 'd' character). The second call to *pat.oneChar* in this example matches the 'd' character in the match sequence, increments the cursor to point at the byte beyond the "cd" string, and then returns and prints "Encountered 'cd'".

As noted earlier, if a sequence of pattern matching commands advances the cursor to the point it "runs off the end" of the character sequence, then the pattern matching sequence fails. The following example demonstrates this (it will print "Failed to match 'cd'"):

```
pat.match( "c" );

  pat.oneChar( 'c' );
  pat.oneChar( 'd' );
  stdout.put( "Encountered 'cd'" nl );

  pat.if_failure

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

pat.endmatch;
```

Note, however, that a pattern matching operation does not fail if it doesn't consume all the characters in the match sequence (that is, it doesn't advance the cursor to the end of the match sequence). The following example succeeds and prints "Encountered 'c'" even though it doesn't consume all the characters in the match sequence:

```
pat.match( "cd" );

  pat.oneChar( 'c' );
  stdout.put( "Encountered 'c'" nl );

  pat.if_failure

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

pat.endmatch;
```

If you're wondering why this shouldn't fail, just note that you can build up complex pattern matching function by making nested and recursive *pat.match* invocations, in such cases you don't want to fail if you've not reached the end of the match sequence because further calls to *pat.match* may handle the remaining characters in the match sequence. In those cases where you really do want to fail if you don't match the entire match sequence, the HLA pattern matching module provides a special function, *pat.EOS*, that explicitly checks for the

end of the match sequence. The following modification to the previous example will display "Failed to match ‘c’":

```
pat.match( "cd" );

  pat.oneChar( ‘c’ );
  pat.EOS();
  stdout.put( "Encountered ‘c’" nl );

  pat.if_failure

  stdout.put( "Failed to match ‘c’" nl );

pat.endmatch;
```

Earlier, this document suggested that a *pat.match..pat.if_failure..pat.endmatch* statement was similar to an IF/ELSE/ENDIF statement insofar as there are two sections of code where you can wind up based on success or failure of the match. In fact, the *pat.match..pat.endmatch* statement is actually closer to an IF/ELSEIF/ELSE/ENDIF statement. If the sequence of pattern matching operations immediately after the *pat.match* statement fail, it is possible to transfer control to another pattern matching operation that will try to succeed. This is known as *alternation* (that is, seeking an alternative match). If the pat.alternate statement appears between the pat.match and the pat.if_failure, then this will supply an alternate pattern matching sequence to try if the main matching sequence fails. Only if both the main and alternate patterns fail will the entire pattern matching operation fail. Consider the following example:

```
pat.match( "cd" );

  pat.oneChar( ‘c’ );
  pat.EOS();
  stdout.put( "Encountered ‘c’" nl );

    pat.alternate

  pat.oneChar( ‘c’ );
  pat.oneChar( ‘d’ );
  pat.EOS();
  stdout.put( "Encountered ‘cd’" nl );

  pat.if_failure

  stdout.put( "Failed to match ‘c’ or ‘cd’" nl );

pat.endmatch;
```

This pattern succeeds and prints "Encountered ‘cd’". It begins by trying to match against ‘c’ (which succeeds) followed by the end of string (which fails). When failure occurs, the *pat.match* statement resets the cursor to the start of the sequence (that is, to the beginning of the "cd" string) and transfers control to the *pat.alternate* statement). This sequence of match operations will match the ‘c’, the ‘d’, and the end of the string, and then print "Encountered ‘cd’".

A *pat.match..pat.endmatch* statement can have any number of *pat.alternate* clauses in it (just as an IF/ELSEIF/ELSE statement can have any number of ELSEIF clauses). The *pat.match* statement will transfer control to the first *pat.alternate* section if the main pattern matching command set fails; it will transfer control to the second *pat.alternate* section if both the main pattern matching sequence and the first alternate sequence fail; a fourth *pat.alternate* section will execute if the main and first two alternate seetions fail; etc. The *pat.if_failure* section will only execute if the main section and all the alternate sections fail to match their patterns.

Note that the *pat.if_failure* section must follow all the *pat.alternate* sections in the *pat.match..pat.endmatch* statement. HLA will report an error if any *pat.alternate* sections follow the *pat.if_failure*. Also remember: the *pat.if_failure* section is not optional. HLA will report an error if a *pat.if_failure* section is not present in a *pat.match..pat.endmatch* statement.

25.3 Pattern Matching Functions Versus User Code

The discussion in the previous section may have led you to believe that a pattern matching section (either the main section or an alternate section) consisted of two parts: the pattern matching code sequence and the user code to execute upon successfully matching the pattern:

```
pat.match( <<character sequence to match>> );

  <<Sequence of match operations>>

  <<Code to execute on a successful match>>

 pat.if_failure

  << Code to execute if the match fails >>

pat.endmatch;
```

In fact, there is no distinction between <<Sequence of match operations>> and <<Code to execute on a successful match>>. The program is going to execute the statements in a matching section until either failure occurs (in which case control transfers to the next *pat.alternate* section, or to the *pat.if_failure* section if there is no alternate), or the execution sequence reaches a *pat.alternate* or *pat.if_failure* statement (at which point control transfers to the first program statement following the *pat.endmatch* clause). The pattern matching functions themselves are really nothing more than 80x86 code that know how to transfer control to some failure clause if the matching function fails. So although most pattern matching statements are organized as described earlier (with the pattern matching operations appearing first and the statements to execute on a successful match occuring afterward), it is possible to inject standard machine instructions and other HLA statements between the pattern matching operations. *However, you must exercise extreme caution when doing so.*

Consider the following example;

```
pat.match( testString );

  pat.oneChar( 'c' );
  stdout.put( "Encountered 'c'" nl );
  pat.EOS();
  stdout.put( "Encountered EOS" nl );

    pat.alternate

  pat.oneChar( 'c' );
  stdout.put( "Encountered 'c'" nl );
  pat.oneChar( 'd' );
  stdout.put( "Encountered 'd'" nl );
  pat.EOS();
  stdout.put( "Encountered EOS" nl );

  pat.if_failure

  stdout.put( "Failed to match 'c' or 'cd' followed by EOS" nl );

pat.endmatch;
```

If testString turns out to have the value "c", then the main matching section succeeds and prints

```
Encountered 'c'
Encountered EOS
```

So far, so good. Now, however, suppose that testString holds the value "cd". In this case, the alternate section succeeds and the program prints the following:

```
Encountered 'c'
Encountered 'c'
Encountered 'd'
Encountered EOS
```

No, this is not a typographical error. Yes, it prints "Encountered 'c'" twice. This happens because the main pattern matching seetion doesn't fail until after it executes the *stdout.put* statement that prints "Encountered 'c'". Generally, failed matches should be *transparent*; that is, they should not affect the system by printing or changing values. This is why most pattern matching sequences appear before any user code (technically called the "semantic action") in a pattern matching sequence. You never want to do something that cannot be undone (such as print data to the console) should the pattern matching operation fail.

25.4 Register and Stack Usage in Pattern Matching Statements

During a pattern matching operation (that is, between the pat.match and pat.endmatch statements), the pattern matching code makes use of most of the 80x86's registers to maintain value such as the cursor, end of sequence pointer, and other values. Therefore, you cannot assume that any register values will be preserved across pattern matching function calls and, even more importantly, you must not play around with the register values between pattern matching function calls as these functions communicate between one another using the registers. Even the stack pointer is not sacrosant. Many pattern matching functions will actually leave data on the stack upon return (to implement a facility known as *backtracking*, which you'll read about a little later). Therefore, you must exercise caution when mixing user statements and pattern matching statements in the same code sequence (that is, this is yet another good reason to put all your "semantic actions" *after* all the pattern matching operations). This section will discuss how the pattern matching code uses registers and and stack, so you can deal with the issue accordingly.

The *pat.match* statement initializes the ESI register with the cursor value (that is, the address of the first character in the match sequence) and EDI with the address of the byte just beyond the end of the match sequence (the *EndOfSequence* value). Whenever a pattern matching function successfully returns, EBX will contain the original cursor value (upon entry into that function) and ESI will contain the new cursor value (that is, it will point beyond all the characters that the function matched). Therefore, EBX..(ESI-1) will be the sequence of characters matched by the function.

Almost all pattern matching functions scramble the value in the EAX register prior to returning (actually, "scramble" is a bad term, most functions actually load the return address for the function into EAX and return by jumping indirectly through EAX's value rather than by executing a RET instruction). Many pattern matching functions modify ECX's value (e.g., for use as a "repeat count" when used with the string instructions). Most of the original Standard Library functions preserve the value held in the EDX register, but because the pattern matching library is extensible, it's dangerous to assume that EDX is preserved.

The *one* register you can count on being preserved is EBP. Upon return from a pattern matching function, you can count on EBP containing the address of your stack frame (assuming you use EBP for this purpose).

As noted earlier, many (most) pattern matching functions do not preserve the value of ESP when they return. In particular, most pattern matching functions actually leave data sitting on the stack when they return. This data may get used by later pattern matching functions should failure occur. Of course, the pattern matching code will eventually clean up after itself; in particular, when you execute the *pat.endmatch* statement the code will clean up the stack and leave it in the same state it was when the program executed the corresponding *pat.match* statement. There are two important implications of this:

You cannot use PUSH and POP instructions to preserve values across a pattern matching function. The following will *not* work:

```
pat.match( "c" );

  push( eax );
  pat.oneChar( 'c' );
  pat.EOS();
  pop( eax );

  pat.if_failure

      stdout.put( "did not match 'c'" nl );

pat.endmatch;
```

The problem is that *pat.oneChar* and *pat.EOS* may leave extra data sitting on the stack when they return. Therefore, the POP(EAX); instruction will not be popping the EAX data originally pushed, instead it will pop off some data left on the stack by the pattern matching functions.

Of course, in a sequence of statements you write, that do not call any pattern matching functions (or anything else that doesn't preserve ESP's value), you may certainly use PUSH and POP in a traditional manner. In particular, if you put all your user code after all the pattern matching function calls, then you can use PUSH and POP to your heart's content. However, be aware that pushing and popping data round pattern matching function calls may not work as you expect.

Because the pat.endmatch clause is responsible for cleaning up the stack, removing any data left on the stack by pattern matching functions, you should never exit out of a *pat.match..pat.endmatch* statement by jumping out of the middle of the code to some label outside the *pat.match..pat.endmatch* sequence. For example, don't do the following:

```
pat.match( "c" );

  push( eax );
  pat.oneChar( 'c' );
  jmp cIsGoodEnough;

  pat.if_failure

      stdout.put( "did not match 'c'" nl );

pat.endmatch;

cIsGoodEnough:// Junk may be left on the stack here.
```

The one exception to this rule is exception-handling code. If an exception occurs in the *pat.match..pat.endmach* statement, the exception handling system will automatically clean up the stack for you before transfering control to your exception handling code sequence. Other than exit by exception, the only way you should leave a pat.match..pat.endmatch statement is by "running off the end" of a pattern matching section (that is, by encountering a *pat.alternate* or *pat.if_failure* clause during the normal sequential execution of the pattern matching section).

One other big piece of advice: avoid using any form of control structures, especially loop control structures, within the pattern matching sequence. In practice, there isn't much need to put a series of pattern matching functions inside a WHILE or FOR loop or inside an IF statement. As you'll discover, the HLA pattern matching module provides a rich variety of functions that automatically process repetitive data or conditionally match one sequence or another (e.g., by using alternation).

Ultimately, the best advice you can follow is to adhere to the original syntax given for the *pat.match..pat.endmatch* statement:

```
pat.match( <<character sequence to match>> );

  << Sequence of match operations>>

  << Code to execute on a successful match >>

 pat.if_failure

  << Code to execute if the match fails >>

pat.endmatch;
```

That is, put all your pattern matching function calls at the beginning of a match section and put the "Code to execute on a successful match" (the "semantic action") after those function calls. Within the semantic action, you can feel free to write any 80x86 code you like (as long as it doesn't make any pattern matching function calls that are part of the current pattern you're matching), use whatever control structures you like, etc. The only restriction is that you shouldn't jump out of the *pat.match..pat.endmatch* statement, as just you were just warned against.

Warning: Do not write a short HLA procedure that contains a sequence of pattern matching function calls that you except to call from within a *pat.match..pat.endmatch* statement. The proper return address may not be sitting on the top of stack when you attempt to return back to the *pat.match..pat.endmatch* statement. and the usual "arrrgh! The stack is messed up!" chaos will ensue. It is possible to write your own pattern matching functions, but they have to be written in a special way. There are instructions on how to do this at the end of this chapter. Although you cannot create a simple procedure in this manner, invoking macros should be okay as long as the expanded text would work properly at the point of the invocation.

25.5 Nesting Pattern Matching Statements

Suppose, using only the *pat.oneChar* pattern matching function, you wanted to match one of the following strings: "c", "cd", "ce", or "cde". You could solve this problem thusly:

```
pat.match( testString );

  pat.oneChar( 'c' );
  pat.EOS();
  stdout.put( "Encountered 'c'" nl );

    pat.alternate

  pat.oneChar( 'c' );
  pat.oneChar( 'd' );
     pat.EOS();
  stdout.put( "Encountered 'cd'" nl );


    pat.alternate

  pat.oneChar( 'c' );
  pat.oneChar( 'e' );
     pat.EOS();
  stdout.put( "Encountered 'ce'" nl );



    pat.alternate

  pat.oneChar( 'c' );
  pat.oneChar( 'd' );
  pat.oneChar( 'e' );
     pat.EOS();
  stdout.put( "Encountered 'cde'" nl );

  pat.if_failure

  stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

pat.endmatch;
```

However, you'll notice that there is a bit of duplicated code here. This makes your program unnecessarily larger and slower. For example, supposed that *testString* holds the value "x". The code above will try the main pattern and all three alternates before failing. Furthermore, note that all of these patterns begin with the character 'c'. Wouldn't it be nice to factor out the test for 'c' and have only a single call to test for this character? Well, as it turns out, this is quite easy to accomplish – *pat.match..pat.endmatch* statements are nestable and recursive, so factoring out subpatterns is fairly easy.

Before discussing how to nest *pat.match* statements, we need to make a quick detour and discuss the *pat.fail* function call. This function does exactly what its name implies: if you execute it within a matching section, that section immediately fails. If you've been told (as you have) not to use control structures like IF/ELSE within the pattern matching code and that you should only use straight-line code sequences, you might wonder about the

purpose of the *pat.fail* function. After all, if some pattern matching sequence contains a call to *pat.fail*, that sequence is always going to fail even if all the functions prior to that point succeed. So why even bother executing the sequence at all? Well, although you should not execute control structures like an IF statement within a pattern matching sequence, don't forget that the pat.match..pat.endmatch is, essentially, an IF/ELSE statement. And, as the title of this subsection suggests, you can nest pat.match statements inside other *pat.match* statements. Therefore, you do have an IF statement – the *pat.match* statement. Consider the following (non-functional) first attempt at using a *pat.match* statement nested inside another to solve the problem given earlier:

```
pat.match( testString );

  pat.oneChar( 'c' );
  pat.match( ???? );

     pat.EOS();
     stdout.put( "matched 'c'" nl );

  pat.alternate;

     pat.oneChar( 'd' );
     pat.match( ????? );

        pat.oneChar( 'e' );
        pat.EOS();
        stdout.put( "matched 'cde'" nl );

     pat.alternate

        pat.EOS();
        stdout.put( "matched 'cd'" nl );

     pat.if_failure

        stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

     pat.endmatch;


  pat.alternate

     pat.oneChar( 'e' );
     pat.EOS();
     stdout.put( "Matched 'ce'" nl );

  pat.if_failure

     stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

  pat.endmatch;

  pat.if_failure

  stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

pat.endmatch;
```

There are two obvious problems with this code sequence. First of all, the easy one: what do we pass the second and third *pat.match* calls? We cannot pass it the original string because we need to pass it a sequence consisting of the characters after the first 'c' that we've already matched. That is, we need to pass this statement the current cursor position (which is in ESI) and the current end of sequence address (which is in EDI). Therefore, we can use the following code to achieve this:

```
pat.match( testString );

  pat.oneChar( 'c' );
  pat.match( esi, edi );

     pat.EOS();
     stdout.put( "matched 'c'" nl );

  pat.alternate;

     pat.oneChar( 'd' );
     pat.match( esi, edi );

        pat.oneChar( 'e' );
        pat.EOS();
        stdout.put( "matched 'cde'" nl );

     pat.alternate

        pat.EOS();
        stdout.put( "matched 'cd'" nl );

     pat.if_failure

        stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

     pat.endmatch;


  pat.alternate

     pat.oneChar( 'e' );
     pat.EOS();
     stdout.put( "Matched 'ce'" nl );

  pat.if_failure

     stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

  pat.endmatch;

  pat.if_failure

  stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

pat.endmatch;
```

The second problem is a bit more difficult to solve. Specifically, we still haven't properly factored out the failure cases. Notice that there are three separate failure cases, all printing the same message. We'd like to have a single failure case than handles everything. As you may have guessed, this is where the *pat.fail* function comes in.

Although you can nest *pat.match* statements, a *pat.match..pat.endmatch* statement, by itself, is not a pattern matching function. It's just a "semantic action" that should appear after all your other pattern matching function calls. However, by the judicial use of the *pat.fail* function, we can turn it into a bonafide pattern matching function. Now a call to *pat.fail* within a pattern matching section isn't going to be very interesting. That's simply going to transfer control to the *pat.match's* associated *pat.if_failure* section. However, what happens if we put the call to *pat.fail* inside the *pat.if_failure* section? The *pat.if_failure* section is not a pattern matching section. If you execute any pattern matching function inside a pat.if_failure section, they will not be processed within that *pat.match..pat.endmatch* statement. Instead, they will be processed by any enclosing *pat.match..pat.endmatch* statement. The following example demonstrates how to use *pat.fail* to simplify the previous code:

```
pat.match( testString );// 1

  pat.oneChar( 'c' );
  pat.match( esi, edi );// 2

     pat.EOS();
     stdout.put( "matched 'c'" nl );

  pat.alternate;

     pat.oneChar( 'd' );
     pat.match( esi, edi );//3

        pat.oneChar( 'e' );
        pat.EOS();
        stdout.put( "matched 'cde'" nl );

     pat.alternate

        pat.EOS();
        stdout.put( "matched 'cd'" nl );

     pat.if_failure

        pat.fail(); // Fails to pat.match #2's if_failure section

     pat.endmatch;


  pat.alternate

     pat.oneChar( 'e' );
     pat.EOS();
     stdout.put( "Matched 'ce'" nl );


  pat.if_failure

     pat.fail();// Fails to pat.match #1's if_failure section

  pat.endmatch;

  pat.if_failure

  stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

pat.endmatch;
```

By the way, you would never actually want to match these four strings this say. There is a pat.matchStr function that provides a much better solution for this problem. Just so you don't walk away thinking these pattern matching functions are terrible, here's a better solution:

```
pat.match( testString );// 1

   pat.matchStr( "c" );
   pat.EOS();
   stdout.put( "matched c" nl );

  pat.alternate
```

```
     pat.matchStr( "cd" );
     pat.EOS();
     stdout.put( "matched cd" nl );

    pat.alternate

     pat.matchStr( "ce" );
     pat.EOS();
     stdout.put( "matched ce" nl );


    pat.alternate

     pat.matchStr( "cde" );
     pat.EOS();
     stdout.put( "matched cde" nl );


    pat.if_failure

    stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

  pat.endmatch;
```

Obviously, this solution is a lot easier to read and understand (and more efficient, too). The previous examples are present to demonstrate nested invocations of the *pat.match* statement.

25.6 Cleanly Nesting Patterns

The previous section demonstrated how to nest patterns and handle the failure case by using the *pat.fail* function. In fact, there are several problems with this approach. In particular, the *pat.match..pat.endmatch* statement is not a pattern matching function (from the perspective of the *pat.match* statement), therefore, for reasons already noted and many unstated, it's not a good idea to use this statement outside the user code ("semantic action") in a pattern matching section. Fortunately, the HLA Standard Library pattern matching module provides a macro that allows you to collect a sequence of pattern matching functions and treat them as though they were a single pattern matching function: the *pat.onePat..pat.endOnePat* statement. The syntax for this statement is the following:

```
pat.onePat;

  <<sequence of pattern matching functions>>

pat.endOnePat;
```

The *pat.onePat..pat.endOnePat* statement is quite similar to the *pat.match..pat.endmatch* statement with three major differences:

There is no *pat.if_failure* section in a *pat.onePat* statement (though pat.alternate sections are perfectly allowable).

You don't pass the match sequence parameter(s) to

pat.onePat – it uses the current cursor and end of sequence pointers.

You generally don't put any user code inside the

pat.onePat..pat.endOnePat sequence (you could, but it's equivalent to putting user statements in the middle of your pattern matching code).

The *pat.onePat* statement can be thought of as a parenthetical pattern matching expression. That is, it groups together a sequence of pattern matching functions and the success of pat.onePat depends entirely upon the success (or failure) of the group of pattern matching statements it encloses. We can use the *pat.onePat* statement to provide another example of a "clean" version of the code in the previous section:

```
static
  index:dword;
```

```
  msg:string[4] :=
            [
                "matched 'c'" nl,
                "matched 'cd'" nl,
                "matched 'ce'" nl,
                "matched 'cde'" nl
            ];

pat.match( testString );// 1

  // Match the leading 'c':

  pat.oneChar( 'c' );
  mov( 0, index );// matched 'c'
  pat.onePat;

        // See if a 'd' follows the 'c':

     pat.oneChar( 'd' );

     // See if an 'e' follows the 'd':

     pat.onePat;

           pat.oneChar( 'e' );
           mov( 3, index );// matched 'cde'

       pat.alternate

           // Note: in the absence of a pattern
           // matching function, this pattern
           // always succeeds.

           mov( 1, index );// matched 'cd'


     pat.endOnePat;

    pat.alternate

     // See if an 'e' follows the 'c':

     pat.oneChar( 'e' );
     mov( 2, index ); // matched 'ce'

  pat.endOnePat;
  pat.EOS();

  mov( index, eax );
  stdout.put( "Matched '", msg[eax*4], "'" nl );

    pat.if_failure

  stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

pat.endmatch;
```

True, this isn't quite as clean as the string example, but you cannot always convert a complex pattern to a few string compares.

Probably the most famous example of a pattern matching sequence is the following, which takes advantage of alternation and parenthetical patterns (i.e., *pat.onePat*):

```
pat.match( someString );


        pat.onePat;

              pat.matchStr( "black" );

           pat.alternate

              pat.matchStr( "blue" );

        pat.endOnePat;
        pat.oneChar( ' ' );
        pat.onePat;

              pat.matchStr( "berry" );

           pat.alternate

              pat.matchStr( "bird" );

        pat.endOnePat;
        stdout.put( "matched" nl );


  pat.if_failure

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

pat.endmatch;
```

This example matches the string "black berry", "blue berry", "black bird", and "blue bird".

25.7 Backtracking

One extremely important facility that the HLA Standard Library pattern matching routines provide is *backtracking*. To understand why backtracking is important, we must expand your pattern matching function repretoire. Up to this point, you've seen *pat.oneChar* that matches exactly one character and *pat.matchStr* that matches a specific string of characters. These functions always match a fixed number of characters (one in the case of *pat.oneChar* and *n* characters, where *n* is the length of the parameter string, in the case of *pat.matchStr*). Some stdlib pattern matching functions, however, match an arbitrary number of characters. For example, consider *pat.oneOrMoreChar*; as its name implies, this function matches one or more occurrences of the same character. That is, a call such as "pat.oneOrMoreChar('a');" will succeed if it can match at least one 'a' character, but it will consume as many 'a' character as it finds in the input stream. The *pat.oneOrMoreChar* eagerly matches characters. That is, it will match as many characters as it finds starting at the cursor position through to the end of the match sequence. Generally, this is desirable for a function with a name like *pat.oneOrMoreChar*, but it can lead to some problems. Consider the following example:

```
pat.match( "aaaa" );

   pat.oneOrMoreChar( 'a' );
   pat.oneChar( 'a' );
   pat.EOS();
   stdout.put( "matched a string of two or more a's" nl );

  pat.if_failure

  stdout.put( "Failed to match a string of two or more a's" nl );
```

```
pat.endmatch;
```

In the absence of backtracking, this example would fail and print the message in the *pat.if_failure* section. This would happen because the *pat.oneOrMoreChar* function would eagerly match all the characters in the match sequence (stopping at the end of the sequence) and the next call to *pat.oneChar* would fail because all of the characters have been consumed. Logically, however, this pattern match should succeed. After all, "aaa" certainly matches the *pat.oneOrMoreChar('a');* function call so there is no reason that this pattern shouldn't succeed. The call to *pat.oneOrMoreChar* should match the first three 'a' characters, the call to *pat.oneChar* should match the fourth, and then the call to *pat.EOS* should match the end of the sequence. In the presence of backtracking, this is exactly what happens.

The HLA Standard Library pattern matching functions that match a variable number of characters all support backtracking. Here's how backtracking works in the previous example:

The *pat.oneOrMoreChar* function eagerly matches as many characters as it can.

The

pat.oneChar attempts to match a single 'a' character. It fails. Control does not immediately transfer to the failure section, however, because the *pat.oneOrMoreChar* function has set up a backtracking frame on the stack (this is the extra stuff that pattern matching functions leave on the stack). In the presence of a backtracking frame on the stack, control transfers back inside the function that pushed the backtracking information (*pat.oneOrMoreChar* in this case).

Inside

pat.oneOrMoreChar, the code backs off one character position, so now it matches only "aaa" rather than "aaaa" and returns as before (still leaving a backtrack frame on the stack, in case it's needed).

Because

pat.oneOrMoreChar has backed up one character at the end of the string, the cursor now points at a single 'a' character, which the *pat.oneChar* function matches.

After

pat.oneChar matches the 'a' character, the cursor is left at the end of the string and the *pat.EOS* function call matches, so the whole statement matches the string.

One area where you can get into big trouble with backtracking is the inclusion of user code ("semantic actions") within the pattern matching code. Because backtracking will cause the reexecution of various instructions within the pattern matching sequence, you can get unexpected results if backtracking occurs. Consider the following example:

```
pat.match( "ccc" );
    pat.oneOrMoreChar( 'c' );
    stdout.put( "Matched first 'c'" nl );
    pat.oneOrMoreChar( 'c' );
    stdout.put( "Matched second 'c'" nl );
    pat.oneOrMoreChar( 'c' );
    stdout.put( "Matched third 'c'" nl );

pat.if_failure

    stdout.put( "failed" nl );

pat.endmatch;
```

This code produces the following output because of backtracking:

```
Matched first 'c'
Matched first 'c'
Matched second 'c'
Matched first 'c'
Matched second 'c'
Matched second 'c'
Matched third 'c'
```

For an explaination of this output, see the section on "Lazy / Eager Evaluation and Pattern Matching Performance" a little later in this document. What's important to realize here is that burying user statements (especially those that affect the outside world, such as output statements) is a very bad idea.

25.8 Pattern Components

Thus far, you've see four different types of pattern objects: parenthetical patterns, characters, strings, and the end of sequence. The HLA Standard Library pattern matching module provides several additional pattern object types. Specifically, the patterns module provides pattern matching functions that test the following:

Character set membership

Characters (case sensitive)

Characters (case insensitive)

Strings (case sensitive)

Strings (case insensitive)

Words (strings delimited by special characters, case sensitive)

Words (case insensitive)

Whitespace

End of string/sequence

Arbitrary character matching

Subpatterns

Cursor position within a match sequence

In addition to these built-in patterns, it is possible for you to extend the pattern matching module by writing your own pattern matching functions. A later section in this document will describe how that is done.

The character and character set pattern matching functions are, by far, the most flexible and powerful of the bunch. Each of these three groups (character sets, case-sensitive characters, and case-insensitive characters) about 20 functions that let you:

Match the character at the cursor position without advancing the cursor (peekCset, peekChar, peekiChar)

Match the character at the cursor position and advance the cursor (oneCset, oneChar, oneiChar).

Match an abitrary number of characters up to the first occurrence of some character (upToCset, upToChar, upToiChar).

Match zero or one characters (zeroOrOneCset, zeroOrOneChar, zeroOrOneiChar).

Match zero or more characters (zeroOrMoreCset, zeroOrMoreChar, zeroOrMoreiChar).

Match one or more characters (oneOrMoreCset, oneOrMoreChar, oneOrMoreiChar).

Match exactly

n characters (firstNCset, exactlyNCset, firstNChar, exactlyNChar, firstNiChar, exactlyNiChar), where *n* is a parameter value.

Match

n or fewer characters (norLessCset, norLessChar, norLessiChar), where *n* is a parameter value.

Match

n or more characters (norMoreCset, norMoreChar, norMoreiChar), where *n* is a parameter value.

Match between

n and *m* characters (ntoMCset, exactlyNtoMCset, ntoMChar, exactlyNtoMChar, ntoMiChar, exactlyNtoMiCar), where *n* and *m* are parameter values.

There are also lazy versions of many of the functions in the above list. We'll discuss the lazy functions in the next section on Eager and Lazy evaluation. As for the specifics of these functions, we'll discuss them in the reference section later in this document.

The important thing to note is that many of these pattern matching function match an arbitrary or parameterized number of characters. For example, a call like the following:

```
pat.exactlyNCset( {'a','b','c'}, 5 );
```

matches exactly five characters and all of them must be members of the set {'a', 'b', 'c'}. The functions that begin with "zeroOrOne…" will either match a single character, or they will succeed without advancing the cursor. The "zeroOrMore…" functions will match as many copies of the character as they can, or they will

succeed without matching any characters. The "oneOrMore…" functions must match at least one character, but will happily match any number of characters afterwards, as well. The "firstN…" functions will match exactly *n* copies of the specified character (set); the "exactlyN…" functions also match exactly *n* characters, but they differ from the "firstN…" functions insofar as the "firstN…" functions don't care what character (if any) appears in the $n+1^{st}$ position. The "exactlyN…" functions require the $n+1^{st}$ character to either be nonexistent (i.e., there were only *n* characters in the string) or it must not be the character (or in the character set) that the function matches. The "norLess…" functions match between zero and *n* copies of a character. The "norMore…' functions match, you guessed it, *n* or more characters in the string. The "nToM…" and "exactlyNtoM…" functions match between *n* and *m* copies of the character in the match sequence; the difference between the two is that the "ntoM…" functions allow the $m+1^{st}$ character to match the pattern whereas the "exactlyNtoM…" functions fail if the $m+1^{st}$ character matches. With all of these functions, it's pretty easy to concoct some pattern matching sequence that can match just about anything.

Though there aren't quite as many string matching functions as there are character and character set functions, there are still a useful variety of functions available. You can match a string (as you've already seen) with the pat.matchStr function. There's a corresponding pat.matchiStr function that does a case insensitive comparison. You can also match all the characters up to (and including) a string with the pat.upToStr function (pat.upToiStr is the case-insensitive version); pat.matchToStr and pat.matchToiStr are similar except they match all characters up to, but not including, the string you pass as a parameter.

There are several other string matching functions you'll want to use. Please consult the reference section at the end of this document for more details on those (especially the whitespace matching functions).

25.9 Lazy / Eager Evaluation and Pattern Matching Performance

Although backtracking is an incredibly useful feature to have, in some very degenerate cases backtracking can produce very slow results. Consider the following example:

```
pat.match( "aaaaaa" )

  pat.zeroOrMoreChar( 'a' );
  pat.oneOrMoreChar( 'a' );
  pat.oneOrMoreChar( 'a' );
  pat.oneOrMoreChar( 'a' );
  pat.oneOrMoreChar( 'a' );
  pat.oneOrMoreChar( 'a' );
  pat.oneOrMoreChar( 'a' );
  stdout.put( "succeeded" nl );

  pat.if_failure

  stdout.put( "failed" nl );

pat.endmatch;
```

Now this particular pattern will succeed. It does so by having the first function match zero characters and all the remaining functions match a single character. This looks simple enough, but if you look closely, you discover that it takes a huge amount of CPU time to match this string. Let's consider what happens here:

The call to pat.zeroOrMoreChar eagerly matches the entire string.

The first call to pat.oneOrMoreChar fails because the first call has consumed all the characters. So backtracking occurs and zeroOrMoreChar releases one character, which the first call to oneOrMoreChar succeeds in matching (this is the second call to that function, by the way).

Control transfers to the second pat.oneOrMoreChar function. It fails because the previous two functions have consumed all the characters in the string. So back tracking occurs. The second call to oneOrMoreChar backtracks to the first call, which tries to give up a character. But when it does, it fails to match, so it back tracks back up to the zeroOrMoreChar call, which backs up a second character and control transfers back to the first oneOrMoreChar call, with the string "aa". The first oneOrMoreChar call matches

both of these characters, so when the call to the second oneOrMoreChar takes place, it fails again. Once again backtracking occurs, this time, however, the first oneOrMoreChar call can give up one character and still

succeed. So control flows back to the second oneOrMoreChar call and it succeeds. Then control falls through to the third oneOrMoreChar call and it fails, and the process starts all over again. To make a (very) long story short, backtracking is going to have exponential worst-case time complexity (that is, it will take on the order of 2^n operations to perform the character match operation.

Though such degenerate cases rarely occur in practice, eager evaluation can be quite expensive when such conditions arise. The solution to this particular problem is to use *lazy evaluation* rather than eager evaluation. For all the functions that match an arbitrary number of characters, there is usually a complementary function that begins with "l_" that performs the same test using lazy evaluation. In the example above, the complementary functions are *pat.l_zeroOrMoreChar* and *pat.l_oneOrMoreChar*. The difference between the eager and the lazy functions is that the eager functions will attempt to match as many characters as the possibly can when first called, and will back off only when backtracking occurs. Lazy functions, on the other hand, will match as few characters as possible and will match more characters only when backtracking occurs. Consider the following rework of the previous example:

```
pat.match( "aaaaaa" )

  pat.l_zeroOrMoreChar( 'a' );
  pat.l_oneOrMoreChar( 'a' );
  pat.l_oneOrMoreChar( 'a' );
  pat.l_oneOrMoreChar( 'a' );
  pat.l_oneOrMoreChar( 'a' );
  pat.l_oneOrMoreChar( 'a' );
  pat.l_oneOrMoreChar( 'a' );
  stdout.put( "succeeded" nl );

  pat.if_failure

  stdout.put( "failed" nl );

pat.endmatch;
```

This function will succeed, just as before, but it won't consume much CPU time at all. The first call matches the minimum number of characters (zero), the remaining functions also match the minimum number of characters (one each), so this code matches the string in one pass without any backtracking.

Lazy evaluation does not completely solve the problem. It is perfectly possible to create a degenerate string that causes lazy evaluation to require exponential time complexity (i.e., run very slow). Indeed, eager evaluation is probably best as the default case. Nonetheless, if you have a good idea of what your match sequences (input strings) will be like, then you can choose eager or lazy evaluation as appropriate to produce the best performance.

In the absense of user code ("semantic actions"), lazy and eager evaluation always produce the same result (even if the performance characteristics are different). That is, if one pattern using eager evaluation matches, then the comparable pattern using lazy evaluation will also match. However, once you embed user statements between the pattern matching functions, the recurring execution of those statements can be greatly affected by your choice of lazy versus eager evaluation. One more reaon to avoid, as much as possible, embedding user instructions in the pattern matching sequences.

Another way to view eager versus lazy evaluation is that eager evaluation always attempts a *maximal match* (matching as many characters as possible) whereas lazy evaluation does a *minimal match* (matching as few characters as possible). In the absence of backtracking, the two approaches could match entirely different strings; but with backtracking present, either method will match a string (though the way they match, and the execution of the associated semantic actions, might be different). If lazy and eager evaluation techniques match a string by matching different substrings during the matching process (that is, if there are two or more ways the code can match the string), we say that the matching operation is *ambiguous*.

25.10 Regular Expressions

If you are familiar with regular expression syntax (e.g., from Unix shell interpreters, various editors, or programs like *grep*), you may find the HLA Standard Library pattern matching routines easier to understand if they are explained in terms of a regular expression syntax. This section will draw some parallels between the HLA Standard Library pattern matching functions and the typical syntaxes that regular expressions use.

In a simple regular expression language, there are two types of characters: *metasymbols* and *alphabetic characters*. Metasymbols have special meaning to the regular expression language and typically include symbols such as '*', '+', '?', '.', '(', ')', and '|'. Alphabetic characters are symbols from a predefined alphabet (an alphabet is simply a set of characters, it isn't necessarily the characters 'a'..'z' from the English alphabet). In most computer systems, the alphabetic is the set of ASCII or UNICODE (UTF-8) characters, sans the metasymbols. For the HLA Standard Library, the alphabet is the set of all 7-bit ASCII characters except the NUL character (ASCII code 0).

In a typical regular expression language (e.g., *grep's* regular expression language), the metasymbols are typically:

```
. ? * + | ( ) [ ] ^ \ ` "
```

The alphabet is the set of all other characters in the system's native character set (e.g., 7-bit ASCII characters). In the event you want to specify one of the metasymbols (which are valid ASCII characters) as standard characters in the alphabet rather than as metasymbols, you can *escape* the meaning of the symbol by prefacing it with a '\' character. For example, the character sequence '*' represents a single asterisk character, '\(' represents a single left parenthesis character, and '\\' represents a single backslash character. When a character has an escape prefix on it, it is treated as any other character in the alphabet.

We can define a regular expression with the following rules:

If *a* is any single character from the alphabet (or an escaped character), then *a* is a regular expression and it matches the single character *a*[1].

If *a* is any single character from the alphabet, then *'a'* is a regular expression and it matches the single character *a*. In many regular expression languages, *a* can actually be a metacharacter (other than ') and quoting the character also escapes it.

If *b* is sequence of zero or more characters from the alphabet, then *"b"* is a regular expression and it matches the string *b*.

The '.' metasymbol represents any character in the character set and is a regular expression. Note the difference between '.' and

a from the previous rule. The *a* represents any single character from the character set whereas '.' is the actual period character. The regular expression *a* matches only the character represented by *a*, the regular expression represented by '.' will match any character in the alphabet.

If *r* is a regular expression and s is a regular expression, then the concatentation of *r*+*s* is also a regular expression and it matches the sequence of characters matched by *r* immediately followed by the sequence of characters matched by *s*. In regular expression terms, this is generally written as *rs*. Note that we may apply this rule recursively to generate strings of any length to match. For example, the string "hello" can be generated as follows:

regex = rs (by definition)

rs = rss (by substituting rs, a regular expression for r)

rss = rsss (by substituting rs for r).

rsss = rssss (by substituting rs for r).

rssss = hello (by substituting 'h' for r, and 'ello' for each of the regular subexpressions ssss, respectively).

If *r* is a regular expression, then *r?* is also a regular expression and it represents zero or one occurrences of *r* (that is, it optionally matches *r*).

If *r* is a regular expression, then *r** is a regular expression and it matches zero or more concatenated occurrences of *r*. Note that *r* can be any regular expression, not just a single character. For example, the regular expression '.*" matches zero or more characters from the alphabet whereas 't*' only matches zero or more 't' characters.

If *r* is a regular expression, then *r*+ is also a regular expression and matches one or more instances of the regular expression *r*. This is actually a shorthand notation for *rr** (that is, one instance of *r* followed by zero or more instances of *r*).

If *r* and *s* are regular expressions, then *r*|*s* is also a regular expression and it will match exactly one occurrence of *r* or *s* (alternation).

---

1. Technically speaking, regular expressions generate strings rather than recognize strings. However, from the theory of computation we can easily show that generation and recognition are equivalent operations, so as this document discusses pattern matching we'll use the term "recognize" or "match" when discussing the behavior of a regular expression.

If *r* is a regular expression, then *(r)* is also a regular expression and it matches the same strings that *r* matches. As for arithmetic expressions, parenthesis are normally used to override precedence and group expessions.

[*charset*] is a regular expression and matches exactly one character from the specified character set. Character sets have the following definition:

A single character *a*, from the alphabet, is a legal character set and the character set *[a]* matches this single character.

A character set of the form

[a-b], where *a* and *b* are both characters in the alphabet with *a's* ordinal value being less than or equal to *b's* ordinal value, is a character set and will match a single character whose value is between *a's* and *b's* ordinal values (inclusive).

If *[c]* and *[d]* are valid character set formulations from items (1) and (2) above, then *[cd]* is a valid character set and matches any character in the union of the two sets *c* and *d*. For example, [a-zA-Z] is the union of [a-z] and [A-Z] and represents the set of all (ASCII/English) alphabetic characters.

If *[c]* is a valid character set, then *[^c]* is also a valid character set and represents the complement of the character set *c*. For example, [^a-zA-Z] represents the set of all non-alphabetic characters (in the ASCII character set, anyway). Note that the "^" symbol must appear immediately after the "[" and this is the only place that the "^" symbol has special meaning.

These few rules are (more than) sufficient to define all regular expressions. Sometimes, however, it is convenient to define a few extra rules to make it easy to specify some complex patterns. In some regular expression languages, for example, an expression of the form *r:[n],* where *r* is a regular expression and *n* is an integer value, will match exactly *n* occurrences of the regular expression *r*. A regular expression of the form *r:[n,m],* where *r* is some regular expression and *n* and *m* are integer values with *n* <= *m* will match between *n* and *m* occurrences of the regular expression *r*.

Here are some common regular expressions and the strings they match:

[a-zA-Z_][a-zA-Z_0-9]*	HLA identifier
[0-9]+	Unsigned integer constant
[0-9] ([_0-9]* [0-9])?	HLA unsigned integer
(\+ | -)? [0-9]+	Signed integer constant
[\+ -]? [0-9]+ (\.[0-9]*)? ([eE] (\+ | -)? [0-9]+)?	Real constant
if	HLA reserved word "if"

Though it would certainly be possible to write some HLA macro that processes regular expressions using the standard syntax given above (for *grep*-like regular expressions), most pattern-matching operations in the HLA Standard Library pattern matching module are accomplished using function calls. This is a bit more typing (and a bit more text to read), but the result is easier to read and understand than a cryptic regular expression, particularly if the regular expression is complex. Of course, the other main difference is that HLA's syntax allows the incorporation of semantic actions (user code to execute on a match), something that traditional regular expression languages do not provide.

It should go without saying, given the number of functions present in the HLA Standard Library pattern matching module, that the stdlib provides a rich set of functions that allow you to process any type of regular expression that you can express using a *grep*-like notation. Let's cover the conversion of *grep*-like regular expressions to HLA Standard Library pattern matching code.

If *a* represents a single character (either a character literal constant or a character variable in HLA), then *pat.oneChar(a);* will succeed if the character at the cursor position matches *a*, it will fail otherwise. Note that other than ‘ and ", HLA does not have any metacharacters. You either supply a character variable or a character constant as the *pat.oneChar* operand.

If *b* represents a string of characters (either a string variable or an HLA literal or manifest string constant) then *pat.matchStr(b);* will succeed if the character sequence at the cursor matches the character string *b*. It fails otherwise.

The HLA stdlib pattern matching module provides several ways to match an arbitrary character. The standard way is with the *pat.skip(n)* function, where n is an unsigned integer. This function succeeds if there are at least *n* characters left in the match sequence string starting at the cursor position. It fails if there are fewer than *n* charcters left in the string. To match a single arbitrary character, you would simply supply the value one as the function's argument: *pat.skip(1);* You could also take the complete of the empty set (which is the entire character set) and pass that to the pat.oneCset function: *pat.oneCset(-{});* Note, however, that the pat.skip function call is more efficient.

Concatentation of two regular expressions is handled by making sequential function calls to the corresponding functions that implement the sub-regular expressions. For example, if function *pat.RRRRR* implements regular expression *r* and funciton *pat.SSSSS* implements regular expression *s*, then the following statements implement the regular expression *rs*:

```
pat.RRRRR( … );
pat.SSSSS( … );
```

Note that you do not have to build string matches up from individual character matches. Just use the pat.matchStr function when matching a sequence of (known) characters.

To match zero or more occurrences of a generic regular expression *r*, the HLA stdlib pattern matching module provides the *pat.zeroOrMorePat…pat.endZeroOrMorePat* statement. You place the statement(s) that implement the regular expression *r* in the body of the *pat.zeroOrMorePat…pat.endZeroOrMorePat* statement and the pattern matching code will attempt to match zero or more occurrences (note that such regular expressions always succeed, as matching zero occurrences is legal).

The HLA stdlib pattern matching module also provides several special case functions that will match zero or more occurrences of:

Any single character (case sensitive or case insensitive)

Any character from a character set

Any white space character

Using these special functions is far more efficient than using the *pat.zeroOrMorePat…pat.endZeroOrMorePat* statement, so you should call these functions if appropriate. For example, to match zero or more alphabetic characters, you'd probably want to use the built-in *pat.zeroOrMoreCset* function thusly:

```
pat.zeroOrMoreCset( {'a'..'z', 'A'..'Z'} );
```

To match one or more occurrences of a generic regular expression *r*, the HLA stdlib pattern matching module provides the *pat.oneOrMorePat…pat.endOneOrMorePat* statement. You place the statement(s) that implement the regular expression *r* in the body of the *pat.oneOrMorePat…pat.endOneOrMorePat* statement and the pattern matching code will attempt to match one or more occurrences. The statement fails if there is not at least one occurrence of the regular expression

The HLA stdlib pattern matching module also provides several special case functions that will match one or more occurrences of:

Any single character (case sensitive or case insensitive)

Any character from a character set

Any white space character

Using these special functions is far more efficient than using the *pat.oneOrMorePat…pat.endOneOrMorePat* statement, so you should call these functions if appropriate. For example, to match an integer value consisting of one or more decimal digits, you'd probably want to use the built-in *pat.oneOrMorePat*function thusly:

```
pat.oneOrMoreCset( {'0'..'9'} );
```

Alternation is handled by the HLA stdlib pattern matching *pat.alternate* statement. For simple regular expressions where the alternation occurs at the outermost level (that is, having the lowest precedence in the regular expression) you can simply use the *pat.alternate* statement within the outermost *pat.match..pat.endmatch* statement. For more complex regular expressions, when the alternation appears inside parenthetical expressions, your best bet is to use the *pat.onePat..pat.alternate..pat.endOnePat* statement to achieve the alternation. the earlier (black|blue)(berry|bird) regular expression example comes to mind here:

```
pat.match( someString );

        pat.onePat;

            pat.matchStr( "black" );

          pat.alternate

            pat.matchStr( "blue" );

        pat.endOnePat;
        pat.oneChar( ' ' );
        pat.onePat;
```

```
                pat.matchStr( "berry" );

            pat.alternate

                pat.matchStr( "bird" );

        pat.endOnePat;
        stdout.put( "matched" nl );


    pat.if_failure

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

pat.endmatch;
```

Parenthetical regular expressions are handled by the *pat.onePat..pat.endOnePat* statement in HLA's pattern matching module. The statements inside this block are executed with higher precedence than the outside code. Consider the following regular expression that matches "blackbird", "bluebird", or "canary":

```
canary | (black|blue) bird
```

Had this been written as "canary | black | blue bird" it wouldn't match the correct strings (it would match "canary", "black", or "blue bird"). Parentheses adjust the precedence of the expression ("|" normally has the lowest precedence of all the regular expression operators, concatenation has very high precedence) to give us the expression we want. To implement the correct regular expression in HLA code, we use the *pat.onePat* and *pat.endOnePat* as our parentheses around the subexpressions:

```
pat.match( someString );

      pat.matchStr( "canary" );

  pat.alternate

      pat.onePat

            pat.matchStr( "black" );

         pat.alternate

            pat.matchStr( "blue" );

      pat.endOnePat;
      pat.matchStr( "bird" );

pat.endmatch;
```

The HLA language provides character sets as a built-in data type, so if you want to match a character set you simply call one of the *pat.*Cset* function and pass a character set as the function's argument. If you want to match against the complement of a character set, you can take the complement by using the set negation ('-') operator, e.g.,

```
// Match non-alpha chars

pat.zeroOrMoreCset( -{'a'..'z', 'A'..'Z'} );
```

See the function reference for a complete description of all the HLA pattern matching functions.

25.11 Pattern Matching Statements

The HLA Standard Library pattern matching module basically breaks up the pattern matching operations into two different categories: statements and functions. Statements are always implemented as macros, functions might be macros or HLA procedures. This section will describe the statements, the next section will describe all the pattern matching functions.

pat.match and pat.endmatch Syntax

The HLA *pat.match* and *pat.endmatch* macros provide the basic tools for pattern matching. These macro statement allow one of the following two syntaxes:

```
// Match syntax #1:

pat.match( StringValue );
  << Sequence of match functions>>
  << Code to execute on a successful match >>

   pat.if_failure

  << Code to execute if the match fails >>

pat.endmatch;
```

StringValue is either an HLA string variable or a string constant.

```
// Match syntax #2:

pat.match( StartOfStr, EndOfStr );
  << Sequence of match functions>>
  << Code to execute on a successful match >>

   pat.if_failure

  << Code to execute if the match fails >>

pat.endmatch;
```

The *StartOfStr* and *EndOfStr* parameters (in syntax #2) must be dword pointers to characters. *StartOfStr* points at the first character of a sequence of characters to match against. *EndOfStr* must point at the first byte *beyond* the last character in the sequence to consider.

The *pat.match* statement, along with many of the matching functions, pushes data onto the stack that may not be cleaned up until execution of the *pat.endmatch* statement. Therefore, you must never jump into a *pat.match..pat.endmatch* block. Likewise, unless you are prepared to clean up the stack yourself, you should not jump out of a *pat.match..pat.endmatch* block[2].

During a normal match operation, the *pat.match* block executes the sequence of string matching functions. If all the functions in the list execute and successfully match their portion of the string, control falls through to the statements after the match sequence. This code should do whatever is necessary if the pattern matches.

On the other hand, if a failure occurs and the pattern matching routines cannot match the specified string, then control transfers to the *pat.if_failure* section and the associated statements execute. Like an IF..THEN..ELSE statement, the program automatically jumps over the *pat.if_failure* section if the "successful match" statements execute.

Consider the following example that matches a string containing a single HLA identifier:

```
pat.match( StrToTest );
```

2. If an exception occurs, the exception handling code will clean up the stack, so exceptions are a legitimate way to prematurely leave a pat.match..pat.endmatch block.

```
    pat.oneCset( { 'a'..'z', 'A'..'Z', '_'} );
    pat.zeroOrMoreCset( { 'a'..'z', 'A'..'Z', '0'..'9', '_'});
    pat.EOS;

    stdout.put( "The string is a valid HLA identifier" nl );

   pat.if_failure

    stdout.put( "The string is not a valid HLA id" nl );

  pat.endmatch;
```

The *pat.oneCset* function matches a single character in *StrToTest* that is a member of the character set appearing in the parameter list. This call requires that the first character of *StrToTest* be an alphabetic character or an underscore.

After *pat.oneCset* matches a character, the pattern matching routines advance a *cursor* into *StrToTest* so that it points just beyond the character matched by *pat.oneCset*. Indeed, all pattern matching routines operate in this manner, they maintain a cursor (in ESI) that points beyond the characters just matched. So had *StrToTest* contained the string "Hello", ESI would be pointing at the "e" in "Hello" immediately after the execution of the *pat.oneCset* pattern matching routine.

The HLA pattern matching routines also return EBX pointing at the first character matched by the routine. In the current example being considered, EBX would be returned pointing at the "H" in "Hello" by the *pat.oneCset* routine.

The *pat.zeroOrMoreCset* routine continues where *pat.oneCset* leaves off. It matches zero or more characters (starting at the location pointed at by ESI). In this particular example, *pat.zeroOrMoreCset* matches zero or more alphanumeric and underscore characters, hence the code will match "ello" in "Hello".

The *pat.EOS* macro matches the end of the string, just to make sure there aren't any other illegal (nonalphanumeric) characters in the string. Note that *pat.zeroOrMoreCset* stops upon encountering the first non-alphanumeric character. The remainder of the pattern (EOS, in this case) must verify that *pat.zeroOrMoreCset* didn't stop on an illegal character.

Had the *StrToTest* variable contained the string "Hello", then the pattern would successfully match the string and the program would print "The string is a valid HLA identifier" and continue execution after the *pat.endmatch* statement.

Because of the way HLA pattern matching routines implement backtracking, each matching routine may leave data on the stack when it successfully returns. This information is necessary to implement backtracking. Although the *pat.endmatch* code cleans up the stack upon exit, it is important to realize that stack is not static. In particular, you cannot push data on the stack before one pattern matching routine and expect to pop it off the stack when that matching routine returns. Instead, you'll pop the data that the matching routine left on the stack (which will probably crash the system if backtracking occurs). It is okay to manipulate the stack in the code section following all the matching functions (or in the failure section), but you must leave the stack intact between calls to pattern matching routines[3].

25.12 Alternation

Another way to handle failure is with the *pat.alternate* macro. A *pat.match..pat.endmatch* macro invocation may optionally contain one or more pat.alternate sections before the (required) *pat.if_failure* section. The *pat.alternate* sections "intercept" failures from the previous section(s) and allow an attempt to rematch the string with a different pattern (somewhat like the ELSEIF clause of an IF..THEN..ELSEIF..ELSE..THEN statement). The following example demonstrates how you could use this:

```
pat.match( StrToTest );

    pat.oneCset( { 'a'..'z', 'A'..'Z', '_'} );
    pat.zeroOrMoreCset( { 'a'..'z', 'A'..'Z', '0'..'9', '_'});
    pat.EOS;
```

3. Note that it is okay to push data onto the stack, do some calculations, and then pop that data off the stack between calls to the pattern matching routines. However, you must ensure that the stack is unchanged since the last pattern matching routine (or since pat.match) or the pattern matching routines will malfunction.

```
    stdout.put( "The string is a valid HLA identifier" nl );

  pat.alternate

    pat.oneOrMoreCset( {'0'..'9', '_'} );
    pat.EOS;

    stdout.puta
    (
        "The string is a valid HLA unsigned integer constant" nl
    );

  pat.if_failure

    stdout.put
    (
        "The string is not a valid HLA id or integer constant" nl
    );

 pat.endmatch;
```

In this example, if the pattern fails to match an HLA identifier, the pattern matching code attempts to see if it matches an integer constant (in the *pat.alternate* section). If this fails as well, then the whole pattern fails to match.

25.13 Pattern Matching Macros

The HLA patterns library implements several of the pattern matching routines as keyword macros within the pat.match macro. These include *pat.EOS*, *pat.position, pat.atPos, pat.skip, pat.getPos, pat.fail, pat.fence, pat.zeroOrOnePat, pat.zeroOrMorePat,* and *pat.oneOrMorePat*. The following sections describe each of these functions.

pat.EOS

```
  pat.match
    << pattern matching statements >>
    pat.EOS;
    // Note that it doesn't make sense to have any more pattern
    // matching statements here because they would never match
    // anything.
  pat.endMatch;
```

The *pat.EOS* macro matches the end of the string. It succeeds if the current "cursor" value (ESI) is pointing at the end of the string to match. It fails otherwise. This macro is great for forcing a string match to consume an entire string. Specifically, by placing a pat.EOS macro invocation at the end of a sequence of pattern matching function calls, you cause the current pattern match to succeed only if the pattern matches the entire string.

pat.position(n)

```
  pat.match
    << pattern matching statements >>
    pat.position( 5 );// Set cursor position to 5, succeed if
                          // match string is at least 5 chars long.
    << pattern matching statements >>
  pat.endMatch;
```

This function repositions the cursor to character *n* in the string that *pat.match* is processing. This function fails if repositioning the cursor would move it outside the bounds of the string. Note that the index of the first character in the string is zero. The macro is great when you need to match a subpattern that begins at some fixed character position within the string.

pat.atPos(n)

```
pat.match
 << pattern matching statements >>
 pat.atPos( 5 );// Succeeds if above matches five characters.
 << pattern matching statements >>
pat.endMatch;
```

This function succeeds if the cursor is currently at position *n* in the string that pat.match is processing. It fails otherwise. This statement is useful when you need to verify that a recursive pattern doesn't exceed some bound in the string.

pat.skip(n)

```
pat.match
 << pattern matching statements >>
 pat.skip( 5 );// Succeeds at least five chars left in
               // match string and advances cursor by
               // five positions.
 << pattern matching statements >>
pat.endMatch;
```

This function advances the cursor *n* positions from its current location. This function succeeds if the new cursor position is within the bounds of the string; it fails otherwise. This function is comparable to matching a specific number of characters in the string. However, this function is much faster than *pat.arb* or one of the character set matching functions.

pat.getPos(var dest:dword)

```
pat.match
 << pattern matching statements >>
 pat.getPos( i );// Succeeds and puts current cursor position
                 // into 'i' variable.
 << pattern matching statements >>
pat.endMatch;
```

This function places the current cursor position in the specified destination operand. This function always succeeds. It does not affect the cursor position. This function stores zero into the *dest* variable if the cursor is at the beginning of the string.

pat.fail

```
pat.match
 << pattern matching statements >>
 pat.onePat
        << pattern matching statements >>
    pat.alternate
        << pattern matching statements >>
        pat.fail;//Always fails if we get to this point.
 pat.endOnePat;
 << pattern matching statements >>

   pat.alternate
 << pattern matching statements >>
pat.endMatch;
```

This forces an immediate failure, backtracking if necessary. This macro is useful for handling exceptional conditions that shouldn't match. That is, if you've matched to some point in the string and you don't want the whole pattern to succeed, executing pat.fail will force an immediate failure. Obviously, this macro invocation only makes sense if alternation is being used in the pattern.

pat.fence

```
pat.match
 << pattern matching statements >>
```

```
pat.fence;// Don't backtrack into previous statements
pat.onePat
    << pattern matching statements >>
pat.endOnePat;
<< pattern matching statements >>

  pat.alternate
<< pattern matching statements >>
pat.endMatch;
```

This function cleans all the backtracking information off the stack. Any pattern matching function following fence will not be able to backtrack to the routines immediately preceding fence in the current *pat.match* statement.

pat.onePat;

```
pat.onePat;
 << pattern matching statements >>
pat.endOnePat;
```

<< *pattern matching statements* >> are some statements that correspond to an HLA pattern sequence (it may contain pattern matching function calls, x86 code, and *pat.alternate* sections; it may not contain a *pat.if_failure* section or a *pat.fence* invocation). The program evaluates the pattern. If it succeeds, control falls to the next statement following the *pat.pattern* call. If it fails, then control transfers directly to the *pat.if_failure* section in the surrounding *pat.match* call.

This macro is primarily used to create "parenthetical patterns" as a convenience when creating complex patterns. Here's an example:

```
pat.match( SomeString );

  pat.onePat

        pat.matchStr( "Black" );

     pat.alternate

        pat.matchStr( "Blue" );

  pat.endOnePat;

  pat.onePat;

        pat.matchStr( "bird" );

     pat.alternate

        pat.matchStr( "berry" );

  pat.endOnePat;

  stdout.put
  (
     "It was 'blackbird', 'bluebird', 'blackberry', or 'blueberry'",
     nl
  );

  pat.if_failure

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

pat.endmatch;
```

Immediately after the *pat.endOnePat* statement, EBX points at the start of the text associated with the pattern match between the *pat.onePat* and *pat.endOnePat* calls. Therefore, you can call functions like *pat.extract* to extract the entire string matched by the pattern between the *pat.onePat* and *pat.endOnePat* calls. This function fully supports backtracking, even across the patterns within the parenthetical pattern expression.

pat.zeroOrOnePat;

```
pat.zeroOrOnePat;
 << pattern matching statements >>
pat.endZeroOrOnePat;
```

<< *pattern matching statements* >> are some statemennts that correspond to an HLA pattern sequence (it may contain pattern matching function calls, x86 code, and *pat.alternate* sections; it may not contain a *pat.if_failure* section or a *pat.fence* invocation). This call invokes the pattern matching function zero or one times to match additional characters in the current string. This function always succeeds since it can match zero times. This function fully supports backtracking.

pat.zeroOrMorePat;

```
pat.zeroOrMorePat;
 << pattern matching statements >>
pat.endZeroOrMorePat
```

Pattern is sequence of pattern matching function calls (just like *pat.pattern* above; including allowing a *pat.alternate* section but not allowing a *pat.if_failure* section). This call invokes the pattern matching function zero or more times to match additional characters in the current string.

pat.oneOrMorePat

```
pat.oneOrMorePat
 << pattern matching statements >>
pat.endOneOrMorePat
```

<< *pattern matching statements* >> are some statemennts that correspond to an HLA pattern sequence (it may contain pattern matching function calls, x86 code, and *pat.alternate* sections; it may not contain a *pat.if_failure* section or a pat.fence invocation). This call invokes the pattern matching function one or more times to match additional characters in the current string. It must match at least one occurrence of the pattern in order to succeed.

25.14 Character Set Matching Functions

The following sections describe each of the character set matching functions provided by the HLA patterns module. These functions take (at the minimum) a character set object. The characters in the match string (at the cursor position) are tested against the characters in the set. These functions succeed if, as appropriate for the specific function, they match characters in the parameterized character set.

You'll notice that there are no "not in character set" type functions in this set. You can easily test to see if a string does not match any characters in a given character set by using the negation of the character set.

procedure pat.peekCset(cst:cset);

```
pat.match
 << pattern matching statements >>
 pat.peekCset( {'0'..'9'} );      // Matches, but does not consume
                                  // a numeric character.
 << pattern matching statements >>
pat.endMatch
```

Succeeds if the following character is in the specified set. Fails otherwise. This function does not affect the cursor position of the match.

procedure pat.oneCset(cst:cset);

```
pat.match
 << pattern matching statements >>
 pat.oneCset( {'0'..'9'} );       // Matches and consumes a numeric char
 << pattern matching statements >>
pat.endMatch
```

Succeeds, and advances the cursor by one position, if the character at the cursor position is in *cst*. Fails otherwise. If this function fails, it does not affect the cursor position.

The following example succeeds and advances the cursor by one position if the character at the current cursor position is an alphabetic character:

```
pat.oneCset( {'a'..'z', 'A'..'Z'} );
```

The following example succeeds and advances the cursor if the current character is *not* an alphabetic character:

```
pat.oneCset( -{'a'..'z', 'A'..'Z'} ); // Note: "-" operator negates cset.
```

procedure pat.upToCset(cst:cset);

```
pat.match
 << pattern matching statements >>
 pat.upToCset( {'0'..'9'} ); // Match all chars up to a numeric char
 << pattern matching statements >>
pat.endMatch;
```

Advances the cursor until it finds a character in *cst*. Fails if none of the characters following the cursor position (to the end of the string) are in *cst*. This advances the cursor position to the character found in the *cst* parameter. Therefore, the next matching function will begin with the character that was present in the set. Note that this function succeeds and skips zero characters if the cursor was pointing at a character in *cst* when this function was called.

This function is great for skipping over some arbitrary number of characters until a character in the given character set is found. If you call the *pat.extract* function immediately after this function call, you'll retrieve the characters skipped over by this function.

```
pat.match
 << pattern matching statements >>
 pat.upToCset( {'0'..'9'} );   // Match all chars up to a numeric char
 pat.extract( s );             // Extract matched chars to 's' string
 << pattern matching statements >>
pat.endMatch;
```

procedure pat.zeroOrOneCset(cst:cset)

```
pat.match
 << pattern matching statements >>
 pat.zeroOrOneCset( {'0'..'9'} );
 << pattern matching statements >>
pat.endMatch;
```

Optionally matches a single character in the string. If the following character is in the character set, this routine advances the cursor and signals success. If the following character is not in the string, this routine simply signals successh without advancing the cursor.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match the character before returning. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. If the following match routine still fails, then this routine fails.

procedure pat.l_ZeroOrOneCset(cst:cset)

```
pat.match
 << pattern matching statements >>
 pat.l_ZeroOrOneCset( {'0'..'9'} );
```

```
    << pattern matching statements >>
   pat.endMatch;
```

Optionally matches a single character in the string. If the following character is in the character set, this routine advances the cursor and signals success. If the following character is not in the string, this routine simply signals success without advancing the cursor.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will start by matching zero characters in the string. If doing so would cause a following match routine to fail, this routine will backtrack and match one character (if possible) and then retry the following match routine. If the following routine still fails, then this routine signals failure.

procedure pat.zeroOrMoreCset(cst:cset);

```
   pat.match
    << pattern matching statements >>
    pat.zeroOrMoreCset( {'0'..'9'} );
    << pattern matching statements >>
   pat.endMatch;
```

Matches zero or more characters in the specified character set. Because this function can match zero characters, it will always succeed. It advances the cursor beyond all the characters that it successfully matches.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up beyond the original cursor position (in which case this routine backtracks to previous functions) or the following match routine(s) succeed.

procedure pat.l_ZeroOrMoreCset(cst:cset);

```
   pat.match
    << pattern matching statements >>
    pat.l_ZeroOrMoreCset( {'0'..'9'} );
    << pattern matching statements >>
   pat.endMatch;
```

Matches zero or more characters in the specified character set. Because this function can match zero characters, it will always succeed. It advances the cursor beyond all the characters that it successfully matches.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine backtracks to previous functions) or the following match routine(s) succeed.

procedure pat.oneOrMoreCset(cst:cset);

```
   pat.match
    << pattern matching statements >>
    pat.oneOrMoreCset( {'0'..'9'} );
    << pattern matching statements >>
   pat.endMatch;
```

Matches one or more characters in the specified character set. Immediately fails if there isn't at least one character in *cst*. It advances the cursor beyond all the characters that it successfully matches.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position plus one (in which case this routine fails) or the following match routine(s) succeed.

procedure pat.l_OneOrMoreCset(cst:cset);

```
   pat.match
    << pattern matching statements >>
    pat.l_OneOrMoreCset( {'0'..'9'} );
    << pattern matching statements >>
```

```
    pat.endMatch;
```

Matches one or more characters in the specified character set. Immediately fails if there isn't at least one character in *cst*. It advances the cursor beyond all the characters that it successfully matches.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., one). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine fails) or the following match routine(s) succeed.

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

```
    pat.match
     << pattern matching statements >>
     pat.exactlyNCset( {'0'..'9'}, 4 );
     << pattern matching statements >>
    pat.endMatch;
```

Matches exactly *n* characters that are members of *cst*. If any of the next *n* characters in the match string are not in *cst*, this routines returns failure. This function advances the cursor by *n* positions if it succeeds.

Note: The character at position (*n*+1) must *not* be a member of *cst* or this routine fails.

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

```
    pat.match
     << pattern matching statements >>
     pat.firstNCset( {'0'..'9'}, 4 );// Matches 4 numeric chars in string
     << pattern matching statements >>
    pat.endMatch;
```

Matches *n* characters that are members of *cst*. On success this function advances the cursor by *n* positions.

Note: The character at position (*n*+1) may be a member of *cst*. Whether or not it is, this routine succeeds if the first *n* characters are members of cst.

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

```
    pat.match
     << pattern matching statements >>
     pat.norLessCset( {'0'..'9'}, i ); // Matches 0..i numeric chars
     << pattern matching statements >>
    pat.endMatch;
```

This routine matches *n* or fewer characters belonging to the *cst* set. This function always succeeds as it can match zero character (which are less than *n* characters).

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string (up to *n*). If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

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

```
    pat.match
     << pattern matching statements >>
     pat.l_NorLessCset( {'0'..'9'}, i ); // Matches 0..i numeric chars
     << pattern matching statements >>
    pat.endMatch;
```

This routine matches *n* or fewer characters belonging to the *cst* set. This function always succeeds as it can match zero character (which are less than *n* characters).

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues

until it advances beyond the end of the string (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

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

```
pat.match
 << pattern matching statements >>
 pat.norMoreCset( {'0'..'9'}, i ); // Matches i..? numeric chars
 << pattern matching statements >>
pat.endMatch;
```

This routine matches at least *n* characters belonging to the *cst* set. If fewer than *n* characters match the set, this routine returns failure. If this function succeeds, it advances the cursor beyond all the characters it matches in *cst*.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

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

```
pat.match
 << pattern matching statements >>
 pat.l_NorMoreCset( {'0'..'9'}, i ); // Matches i..? numeric chars
 << pattern matching statements >>
pat.endMatch;
```

This routine matches at least *n* characters belonging to the *cst* set. If fewer than *n* characters match the set, this routine returns failure. If this function succeeds, it advances the cursor beyond all the characters it matches in *cst*.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., *n*). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

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

```
pat.match
 << pattern matching statements >>
 pat.ntoMCset( {'0'..'9'}, i, j ); // Matches i..j numeric chars
 << pattern matching statements >>
pat.endMatch;
```

This routine matches at least *n* characters and no more than *m* characters belonging to the cst set. If fewer than *n* characters match the set, this routine returns failure. This routine does not fail if more than *m* characters belong to the set. However, it only matches through position *m*.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

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

```
pat.match
 << pattern matching statements >>
 pat.l_NtoMCset( {'0'..'9'}, i, j ); // Matches i..j numeric chars
 << pattern matching statements >>
pat.endMatch;
```

This routine matches at least *n* characters and no more than *m* characters belonging to the cst set. If fewer than *n* characters match the set, this routine returns failure. This routine does not fail if more than *m* characters belong to the set. However, it only matches through position *m*.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., n). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond position m (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

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

```
pat.match
 << pattern matching statements >>
 pat.exactlyNtoMCset( {'0'..'9'}, i, j ); // Matches i..j numeric chars
 << pattern matching statements >>
pat.endMatch;
```

This routine matches at least *n* characters and no more than *m* characters belonging to the *cst* set. If fewer than *n* characters match the set, this routine returns failure. This routine fails if more than *m* characters belong to the set.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

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

```
pat.match
 << pattern matching statements >>
 pat.l_ExactlyNtoMCset( {'0'..'9'}, i, j );        // Matches i..j
                                                   //  numeric chars
 << pattern matching statements >>
pat.endMatch;
```

This routine matches at least *n* characters and no more than m characters belonging to the cst set. If fewer than *n* characters match the set, this routine returns failure. This routine fails if more than *m* characters belong to the set.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., n). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond position m (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

25.15 Character Matching Functions

The following sections describe each of the character matching functions provided by the HLA patterns module. These functions take (at the minimum) a character object. The characters in the match string (at the cursor position) are tested against the character. These functions succeed if, as appropriate for the specific function, they match the parameterized character.

You'll notice that there are no "not equal to character " type functions in this group. You can easily test to see if a string does not match any character using the character set pattern matching functions.

```
procedure pat.peekChar( c:char );
```

```
pat.match
 << pattern matching statements >>
 pat.peekChar( 'a' ); // Matches 'a'
 << pattern matching statements >>
pat.endMatch;
```

This routine succeeds if the character pointed at by the cursor (ESI) is equal to *c*; it fails otherwise. This routine does not advance the cursor if it succeeds.

```
procedure pat.oneChar( c:char );
```

```
pat.match
 << pattern matching statements >>
 pat.oneChar( 'a' ); // Matches 'a'
 << pattern matching statements >>
pat.endMatch;
```

This routine succeeds if the character pointed at by the cursor (ESI) is equal to *c*; it fails otherwise. If it succeeds, this routine advances the cursor over the character it matches.

```
procedure pat.upToChar( c:char );
```

```
pat.match
 << pattern matching statements >>
 pat.oneChar( ch ); // Matches char in ch register
 << pattern matching statements >>
pat.endMatch;
```

This routine matches all characters in a string from the cursor position up to the specified parameter. It fails if the specified character is not in the string. Note that this routine leaves the cursor pointing at the character specified by the parameter (i.e., it still remains to be matched). A call to pat.extract immediately after this function will create a string with all the characters up to, but not including, the character passed as the parameter.

```
procedure pat.zeroOrOneChar( c:char );
```

```
pat.match
 << pattern matching statements >>
 pat.zeroOrOneChar( c ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This routine matches zero or one occurrences of the character parameter. Because it can match zero characters, this function always succeeds. This function is great for matching an optional character in a pattern.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match one character in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine.

```
procedure pat.l_ZeroOrOneChar( c:char );
```

```
pat.match
 << pattern matching statements >>
 pat.zeroOrOneChar( c ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This routine matches zero or one occurrences of the character parameter. In other words, it lets you check for the presence of an optional character. Because this function can match zero characters, it always succeeds.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. If that fails, then this routine fails.

```
procedure pat.zeroOrMoreChar( c:char );
```

```
pat.match
 << pattern matching statements >>
 pat.zeroOrMoreChar( c ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This routine matches zero or more occurrences of the character parameter. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. Because this function can match zero characters, it always succeeds.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine fails) or the following match routine(s) succeed.

procedure pat.l_ZeroOrMoreChar(c:char);

```
pat.match
 << pattern matching statements >>
 pat.l_ZeroOrMoreChar( c ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This routine matches zero or more occurrences of the character parameter. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. Because this function can match zero characters, it always succeeds.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine fails) or the following match routine(s) succeed.

procedure pat.oneOrMoreChar(c:char);

```
pat.match
 << pattern matching statements >>
 pat.oneOrMoreChar( c ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This routine matches one or more occurrences of the character parameter. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. It fails if there isn't at least one copy of c at the cursor position.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the cursor position of the first character it matched (in which case this routine fails) or the following match routine(s) succeed.

procedure pat.l_OneOrMoreChar(c:char);

```
pat.match
 << pattern matching statements >>
 pat.l_OneOrMoreChar( c ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This routine matches one or more occurrences of the character parameter. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. It fails if there isn't at least one copy of *c* at the cursor position.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., one). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case following functions fail and this function returns the failure on back up the invocation chain) or the following match routine(s) succeed.

procedure pat.exactlyNChar(c:char; n:uns32);

```
pat.match
 << pattern matching statements >>
 pat.exactlyNChar( c, n ); // Matches char in c variable
 << pattern matching statements >>
```

```
    pat.endMatch;
```

This routine matches exactly *n* copies of the character *c* in the string. If more, or less, copies of *c* appear in the string at the current cursor position then this routine fails. Note that the character at cursor position (*n*+1) must not be equal to *c* or this function fails even if the first *n* characters do match.

procedure pat.firstNChar(c:char; n:uns32);

```
    pat.match
     << pattern matching statements >>
     pat.firstNChar( c, n ); // Matches char in c variable
     << pattern matching statements >>
    pat.endMatch;
```

This routine matches *n* copies of the character *c* in the string. If fewer than *n* copies of *c* appear in the string, this routine fails. If more copies of *c* appear in the string, this routine succeeds, however, it only matches the first *n* copies.

procedure pat.norLessChar(c:char; n:uns32);

```
    pat.match
     << pattern matching statements >>
     pat.norLessChar( c, n ); // Matches char in c variable
     << pattern matching statements >>
    pat.endMatch;
```

This procedure matches *n* or fewer copies of *c* in the current string. If additional copies of *c* appear in the string, this routine still succeeds but it only matches the first n copies.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine fails) or the following match routine(s) succeed.

procedure pat.l_NorLessChar(c:char; n:uns32);

```
    pat.match
     << pattern matching statements >>
     pat.l_NorLessChar( c, n ); // Matches char in c variable
     << pattern matching statements >>
    pat.endMatch;
```

This procedure matches *n* or fewer copies of *c* in the current string. If additional copies of *c* appear in the string, this routine still succeeds but it only matches the first *n* copies.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine passes back the failure returned by the following matching functions) or the following match routine(s) succeed.

procedure pat.norMoreChar(c:char; n:uns32);

```
    pat.match
     << pattern matching statements >>
     pat.norMoreChar( c, n ); // Matches char in c variable
     << pattern matching statements >>
    pat.endMatch;
```

This procedure matches *n* or more copies of *c* in the current string. It fails if there are fewer than *n* copies of *c*.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to position n (in which case this routine fails) or the following match routine(s) succeed.

procedure pat.l_NorMoreChar(c:char; n:uns32);

```
pat.match
 << pattern matching statements >>
 pat.l_NorMoreChar( c, n ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This procedure matches n or more copies of *c* in the current string. It fails if there are fewer than *n* copies of character *c* in the string.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., n). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine returns the failure reported by the following match routines) or the following match routine(s) succeed.

procedure pat.ntoMChar(c:char; n:uns32; m:uns32);

```
pat.match
 << pattern matching statements >>
 pat.ntoMChar( c, n, m );
 << pattern matching statements >>
pat.endMatch;
```

This procedure matches between *n* and m copies of the character *c* starting at the current cursor (ESI) position. This routine succeeds even if there are more than *m* copies of the character, however, it will only match the first *m* characters in the string.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to position n (in which case this routine fails) or the following match routine(s) succeed.

procedure pat.l_NtoMChar(c:char; n:uns32; m:uns32);

```
pat.match
 << pattern matching statements >>
 pat.l_NtoMChar( c, n, m );
 << pattern matching statements >>
pat.endMatch;
```

This procedure matches between *n* and *m* copies of the character *c* starting at the current cursor (ESI) position. This routine succeeds even if there are more than *m* copies of the character, however, it will only match the first *m* characters in the string.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., n). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond position m (in which case this routine returns the failure) or the following match routine(s) succeed.

procedure pat.exactlyNtoMChar(c:char; n:uns32; m:uns32);

```
pat.match
 << pattern matching statements >>
 pat.exactlyNtoMChar( c, n, m );
 << pattern matching statements >>
pat.endMatch;
```

This procedure matches between *n* and *m* copies of the character *c* starting at the current cursor (ESI) position. This routine fails if there are more than *m* copies (or fewer than *n* copies) of the character in the string.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to position n (in which case this routine fails) or the following match routine(s) succeed.

procedure pat.l_ExactlyNtoMChar(c:char; n:uns32; m:uns32);

```
pat.match
 << pattern matching statements >>
 pat.l_ExactlyNtoMChar( c, n, m );
 << pattern matching statements >>
pat.endMatch;
```

This procedure matches between *n* and *m* copies of the character *c* starting at the current cursor (ESI) position. This routine fails if there are more than *m* copies (or fewer than *n* copies) of the character in the string.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., n). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond position m (in which case this routine fails) or the following match routine(s) succeed.

25.16 Case Insensitive Character Matching Routines

These routines are semantically identical to the above routines with one difference- when they compare the characters they use a case insensitive comparison. Please see the descriptions above for an explanation of these routines.

procedure pat.peekiChar(c:char);

```
pat.match
 << pattern matching statements >>
 pat.peekChar( 'a' ); // Matches 'a'
 << pattern matching statements >>
pat.endMatch;
```

This routine succeeds if the character pointed at by the cursor (ESI) is equal to *c* using a case insenstive comparison; it fails otherwise. This routine does not advance the cursor if it succeeds.

procedure pat.oneiChar(c:char);

```
pat.match
 << pattern matching statements >>
 pat.oneChar( 'a' ); // Matches 'a'
 << pattern matching statements >>
pat.endMatch;
```

This routine succeeds if the character pointed at by the cursor (ESI) is equal to *c* using a case insenstive comparison; it fails otherwise. If it succeeds, this routine advances the cursor over the character it matches.

procedure pat.upToiChar(c:char);

```
pat.match
 << pattern matching statements >>
 pat.oneChar( ch ); // Matches char in ch register
 << pattern matching statements >>
pat.endMatch;
```

This routine matches all characters, using a case insenstive comparison, in a string from the cursor position up to the specified parameter. It fails if the specified character is not in the string. Note that this routine leaves the cursor pointing at the character specified by the parameter (i.e., it still remains to be matched). A call to pat.extract immediately after this function will create a string with all the characters up to, but not including, the character passed as the parameter.

procedure pat.zeroOrOneiChar(c:char);

```
pat.match
 << pattern matching statements >>
 pat.zeroOrOneChar( c ); // Matches char in c variable
 << pattern matching statements >>
```

```
pat.endMatch;
```

This routine matches zero or one occurrences of the character parameter using a case insenstive comparison. Because it can match zero characters, this function always succeeds. This function is great for matching an optional character in a pattern.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match one character in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine.

procedure pat.l_ZeroOrOneiChar(c:char);

```
pat.match
 << pattern matching statements >>
 pat.zeroOrOneChar( c ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This routine matches zero or one occurrences of the character parameter using a case insenstive comparison. In other words, it lets you check for the presence of an optional character. Because this function can match zero characters, it always succeeds.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. If that fails, then this routine fails.

procedure pat.zeroOrMoreiChar(c:char);

```
pat.match
 << pattern matching statements >>
 pat.zeroOrMoreChar( c ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This routine matches zero or more occurrences of the character parameter using a case insenstive comparison. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. Because this function can match zero characters, it always succeeds.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine fails) or the following match routine(s) succeed.

procedure pat.l_ZeroOrMoreiChar(c:char);

```
pat.match
 << pattern matching statements >>
 pat.l_ZeroOrMoreChar( c ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This routine matches zero or more occurrences of the character parameter using a case insenstive comparison. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. Because this function can match zero characters, it always succeeds.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine fails) or the following match routine(s) succeed.

procedure pat.oneOrMoreiChar(c:char);

```
pat.match
 << pattern matching statements >>
 pat.oneOrMoreChar( c ); // Matches char in c variable
 << pattern matching statements >>
```

```
pat.endMatch;
```

This routine matches one or more occurrences of the character parameter using a case insenstive comparison. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. It fails if there isn't at least one copy of c at the cursor position.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the cursor position of the first character it matched (in which case this routine fails) or the following match routine(s) succeed.

procedure pat.l_OneOrMoreiChar(c:char);

```
pat.match
 << pattern matching statements >>
 pat.l_OneOrMoreChar( c ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This routine matches one or more occurrences of the character parameter using a case insenstive comparison. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. It fails if there isn't at least one copy of *c* at the cursor position.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., one). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case following functions fail and this function returns the failure on back up the invocation chain) or the following match routine(s) succeed.

procedure pat.exactlyNiChar(c:char; n:uns32);

```
pat.match
 << pattern matching statements >>
 pat.exactlyNChar( c, n ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This routine matches exactly *n* copies of the character *c* in the string using a case insenstive comparison. If more, or less, copies of *c* appear in the string at the current cursor position then this routine fails. Note that the character at cursor position (*n*+1) must not be equal to *c* or this function fails even if the first *n* characters do match.

procedure pat.firstNiChar(c:char; n:uns32);

```
pat.match
 << pattern matching statements >>
 pat.firstNChar( c, n ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This routine matches *n* copies of the character *c* in the string using a case insenstive comparison. If fewer than *n* copies of *c* appear in the string, this routine fails. If more copies of *c* appear in the string, this routine succeeds, however, it only matches the first *n* copies.

procedure pat.norLessiChar(c:char; n:uns32);

```
pat.match
 << pattern matching statements >>
 pat.norLessChar( c, n ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This procedure matches *n* or fewer copies of *c* in the current string using a case insenstive comparison. If additional copies of *c* appear in the string, this routine still succeeds but it only matches the first n copies.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine fails) or the following match routine(s) succeed.

procedure pat.l_NorLessiChar(c:char; n:uns32);

```
pat.match
 << pattern matching statements >>
 pat.l_NorLessChar( c, n ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This procedure matches *n* or fewer copies of *c* in the current string using a case insenstive comparison. If additional copies of *c* appear in the string, this routine still succeeds but it only matches the first *n* copies.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine passes back the failure returned by the following matching functions) or the following match routine(s) succeed.

procedure pat.norMoreiChar(c:char; n:uns32);

```
pat.match
 << pattern matching statements >>
 pat.norMoreChar( c, n ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This procedure matches *n* or more copies of *c* in the current string using a case insenstive comparison. It fails if there are fewer than *n* copies of *c*.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to position n (in which case this routine fails) or the following match routine(s) succeed.

procedure pat.l_NorMoreiChar(c:char; n:uns32);

```
pat.match
 << pattern matching statements >>
 pat.l_NorMoreChar( c, n ); // Matches char in c variable
 << pattern matching statements >>
pat.endMatch;
```

This procedure matches n or more copies of *c* in the current string using a case insenstive comparison. It fails if there are fewer than *n* copies of character *c* in the string.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., n). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine returns the failure reported by the following match routines) or the following match routine(s) succeed.

procedure pat.ntoMiChar(c:char; n:uns32; m:uns32);

```
pat.match
 << pattern matching statements >>
 pat.ntoMChar( c, n, m );
 << pattern matching statements >>
pat.endMatch;
```

This procedure matches between *n* and m copies of the character *c* starting at the current cursor (ESI) position, using a case insenstive comparison. This routine succeeds even if there are more than *m* copies of the character, however, it will only match the first *m* characters in the string.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to position n (in which case this routine fails) or the following match routine(s) succeed.

procedure pat.l_NtoMiChar(c:char; n:uns32; m:uns32);

```
pat.match
 << pattern matching statements >>
 pat.l_NtoMChar( c, n, m );
 << pattern matching statements >>
pat.endMatch;
```

This procedure matches between *n* and *m* copies of the character *c* starting at the current cursor (ESI) position, using a case insenstive comparison. This routine succeeds even if there are more than *m* copies of the character, however, it will only match the first *m* characters in the string.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., n). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond position m (in which case this routine returns the failure) or the following match routine(s) succeed.

procedure pat.exactlyNtoMiChar(c:char; n:uns32; m:uns32);

```
pat.match
 << pattern matching statements >>
 pat.exactlyNtoMChar( c, n, m );
 << pattern matching statements >>
pat.endMatch;
```

This procedure matches between *n* and *m* copies of the character *c* starting at the current cursor (ESI) position, using a case insenstive comparison. This routine fails if there are more than *m* copies (or fewer than *n* copies) of the character in the string.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to position n (in which case this routine fails) or the following match routine(s) succeed.

procedure pat.l_ExactlyNtoMiChar(c:char; n:uns32; m:uns32);

```
pat.match
 << pattern matching statements >>
 pat.l_ExactlyNtoMChar( c, n, m );
 << pattern matching statements >>
pat.endMatch;
```

This procedure matches between *n* and *m* copies of the character *c* starting at the current cursor (ESI) position, using a case insenstive comparison. This routine fails if there are more than *m* copies (or fewer than *n* copies) of the character in the string.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., n). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond position m (in which case this routine fails) or the following match routine(s) succeed.

String Matching Functions

procedure pat.matchStr(s:string);

```
pat.match
 << pattern matching statements >>
 pat.matchStr( someString );
 << pattern matching statements >>
pat.endMatch;
```

If the sequence of characters at the current cursor position (ESI) match the specified string, this routine succeeds, otherwise it fails. Note that additional characters may appear in the match string after the characters matched by *s*.

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

```
pat.match
 << pattern matching statements >>
 pat.matchiStr( someString );
 << pattern matching statements >>
pat.endMatch;
```

Like *pat.matchStr*, except this routine does a case insensitive comparison.

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

```
pat.match
 << pattern matching statements >>
 pat.matchToStr( someString );
 << pattern matching statements >>
pat.endMatch;
```

This routine matches all characters up to, and including, the parameter string *s*. If it matches a string and a following pattern matching routine fails, this routine handles the backtracking and searches for the next string that matches. The backtracking is lazy insofar is this routine will always match the minimum number of characters up to *s* in the string in order to succeed. When backtracking occurs, this function will skip over the string it has matched and search for another occurrence. This function will fail if it cannot find another occurrence of *s* in the match string.

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

```
pat.match
 << pattern matching statements >>
 pat.upToStr( someString );
 << pattern matching statements >>
pat.endMatch;
```

This routine matches all characters up to, but not including, the parameter string *s*. If it matches a string and a following pattern matching routine fails, this routine handles the backtracking and searches for the next string that matches. The backtracking is lazy insofar is this routine will always match the minimum number of occurrences of *s* in the string in order to succeed.

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

```
pat.match
 << pattern matching statements >>
 pat.matchToiStr( someString );
 << pattern matching statements >>
pat.endMatch;
```

Like *pat.matchToStr*, except this routine does a case insensitive comparison.

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

```
pat.match
 << pattern matching statements >>
 pat.upToiStr( someString );
 << pattern matching statements >>
pat.endMatch;
```

Like *pat.upToStr*, except this routine does a case insensitive comparison.

```
procedure pat.matchWord( s:string );
```

```
pat.match
 << pattern matching statements >>
```

```
    pat.matchWord( someString );
    << pattern matching statements >>
   pat.endMatch;
```

This routine is similar to *pat.matchStr* except that it requires a delimiter character after the string it matches. The delimiter character is a member of the *WordDelims* character set (internal to the patterns.hhf code). *WordDelims* is, by default, the character set "-{'a'..'z', 'A'..'Z', '0'..'9', '_'}" (that is, all character except the alphanumeric characters and the underscore). See the *getWordDelims* and *setWordDelims* procedures if you are interested in changing the word delimiters set.

```
procedure pat.matchiWord( s:string );
```

```
    pat.match
     << pattern matching statements >>
     pat.matchiWord( someString );
     << pattern matching statements >>
    pat.endMatch;
```

Just like *pat.matchWord*, except this routine does a case insensitive comparison.

```
procedure pat.getWordDelims( var cst:cset );
```

```
    pat.getWordDelims( destinationCSet );
```

This function makes a copy of the internal *WordDelims* character set and places this copy in the specified cst parameter. Note that you do not have to call this function inside a *pat.match..pat.endMatch* sequence (though it's perfectly okay to do so).

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

```
    pat.setWordDelims( newDelimsCSet );
```

This function stores the value of the *cst* character set into the *WordDelims* character set. This allows you to change the *WordDelims* character set to your liking. Note that you do not have to call this function inside a *pat.match..pat.endMatch* sequence (though it's perfectly okay to do so).

25.17 String Extraction Functions

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

```
    pat.match
     << pattern matching statements >>
     pat.extract( someAllocatedStringObject );
     << pattern matching statements >>
    pat.endMatch;
```

Whenever a pattern matching routine successfully matches zero or more characters in the string, the pattern matching routine returns a pointer to the start of the matched characters in EBX and a pointer to the position just beyond the last matched position in ESI. You may use the *pat.extract* procedure to create an HLA-compatible string of these matched characters. This routine will raise an exception if the destination string isn't big enough to hold the extracted characters.

Note that *pat.extract* will only extract those characters that the immediately previous string matching function matched. If you want to extract a string from a sequence of match functions, use the *pat.onePat..pat.endOnePat* sequence to group the functions whose matched string you want to extract.

Be careful about making calls to *pat.extract* when backtracking can occur. Though *pat.extract* will work fine in the event of backtracking, you will take a big performance hit if the system has to make a copy of the same string over and over again if backtracking occurs frequently.

Warning: *pat.extract* should only be called in the "success" section of a *pat.match..pat.endmatch* block. Any other invocation could create a problem. In general, you must ensure that EBX and ESI point at reasonable spots within the same string. Note that pattern match failure does not guarantee that EBX contains a reasonable value. Therefore, you should not use *pat.extract* at a point where string failure could have occurred unless you explicitly set up EBX (and, possibly, ESI) yourself.

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

```
pat.match
 << pattern matching statements >>
 pat.a_extract();
 mov( eax, stringVariable );
 << pattern matching statements >>
pat.endMatch;
```

Whenever a pattern matching routine successfully matches zero or more characters in the string, the pattern matching routine returns a pointer to the start of the matched characters in EBX and a pointer to the position just beyond the last matched position in ESI. You may use the pat.a_extract procedure to create an HLA-compatible string of these matched characters. *pat.a_extract* will allocate storage for the string on the heap, copy the matched characters to this string, and then store a pointer to the new string in the string variable passed as a reference parameter to *pat.a_extract*.

Be careful about making calls to *pat.a_extract* when backtracking can occur. Though *pat.a_extract* will work fine in the event of backtracking, you will take a big performance hit if the system has to make a copy of the same string over and over again if backtracking occurs frequently. Also, note that unless you take care to free the string data allocated on a previous call to *pat.a_extract* (before the backtracking occurs), you'll wind up with a "memory leak". For this reason, you should use *pat.extract* on a preallocated string rather than calling *pat.a_extract* to allocate the string.

Warning: *pat.a_extract* should only be called in the "success" section of a *pat.match..pat.endmatch* block. Any other invocation could create a problem. In general, you must ensure that EBX and ESI point at reasonable spots within the same string. Note that pattern match failure does not guarantee that EBX contains a reasonable value. Therefore, you should not use *pat.a_extract* at a point where string failure could have occurred unless you explicitly set up EBX (and, possibly, ESI) yourself.

25.18 Whitespace and End of String Matching Functions

These convenient routines match a sequence of whitespace characters as well as the end of the current string. By default, these routines assume that whitespace consists of all the control characters, the ASCII space (#$20), and the del code (#$7f). You can change this definition using the *pat.getWhiteSpace* and *pat.setWhiteSpace* procedures.

```
procedure pat.getWhiteSpace( var cst:cset );
```

```
pat.getWhiteSpace( destinationCSet );
```

This function returns the current value of the internal *WhiteSpace* character set. It stores the result in the reference parameter *cst*. Note that you do not have to call this function inside a *pat.match..pat.endMatch* sequence (though it's perfectly okay to do so).

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

```
pat.setWhiteSpace( newCSet );
```

This procedure copies the specified character set to the internal *WhiteSpace* character set. All future whitespace matching procedures will use this new value when matching white space characters. Note that you do not have to call this function inside a *pat.match..pat.endMatch* sequence (though it's perfectly okay to do so).

```
procedure pat.zeroOrMoreWS;
```

```
pat.match
 << pattern matching statements >>
 pat.zeroOrMoreWS();
 << pattern matching statements >>
pat.endMatch;
```

This routine matches zero more more whitespace characters. This routine uses an "eager" matching algorithm.

procedure pat.oneOrMoreWS;

```
pat.match
 << pattern matching statements >>
 pat.oneOrMoreWS();
 << pattern matching statements >>
pat.endMatch;
```

This routine matches zero more more whitespace characters. This routine uses an "eager" matching algorithm; it will backtrack over matched white space characters at the end if the following match functions require some whitespace characters to succeed. If there isn't at least one whitespace character at the cursor position, this function fails. Otherwise, it succeeds.

procedure pat.WSorEOS;

```
pat.match
 << pattern matching statements >>
 pat.WSorEOS();
 << pattern matching statements >>
pat.endMatch;
```

This routine matches a single whitespace character or the end of the string. It fails if there are characters left in the string and the character at the cursor position is not a white space character.

procedure pat.WSthenEOS;

```
pat.match
 << pattern matching statements >>
 pat.WSthenWS();
 << pattern matching statements >>
pat.endMatch;
```

This routine matches zero or more white space characters that appear at the end of the current string. It fails if there are any other characters before the end of the string.

procedure pat.peekWS;

```
pat.match
 << pattern matching statements >>
 pat.peekWS();
 << pattern matching statements >>
pat.endMatch;
```

This routine succeeds if the next character in the string is a whitespace character. However, it does not advance the cursor over the character.

procedure pat.peekWSorEOS;

```
pat.match
 << pattern matching statements >>
 pat.peekWSorEOS();
 << pattern matching statements >>
pat.endMatch;
```

This routine succeeds if the next character in the string is a white space character or if there are no more characters in the string. It does not advance the cursor.

25.19 Matching an Arbitrary Sequence of Characters

procedure pat.arb;

```
pat.match
 << pattern matching statements >>
```

```
    pat.arb();
    << pattern matching statements >>
  pat.endMatch;
```

This routine matches zero or more characters. It uses an "aggressive" or "eager" matching algorithm, immediately matching all the remaining characters in the string. If following matching routines fail, this routine backtracks one character at a time until reaching the initial starting position (in which case this routine fails) or the following matching routine(s) succeed.

```
procedure pat.l_arb; external;
```

```
  pat.match
    << pattern matching statements >>
    pat.l_arb();
    << pattern matching statements >>
  pat.endMatch;
```

This is a "lazy" or "deferred" version of the above routine. It matches zero characters and succeeds; if a following match routine fails, this routine backtracks by advancing the cursor one position for each failure. If this routine advances beyond the end of the string during backtracking, it reports failure.

25.20 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 backtracking. Generally, pattern matching functions that can match strings of varying lengths should support backtracking[4]. 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 backtracking 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 maximum 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.

If function that support 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 back tracking returns

4. 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.

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[5]. 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 the following matching functions to succeed, choosing lazy evaluation will run much more slowly than eager evaluation. 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 string 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)[6].

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;
```

5. 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.

6. The HLA Standard Library Pattern Matching routines preserve EDX, so this is probably a good convention to follow so you don't surprise your users.

```
    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;
```

If your function needs to support backtracking, 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[7]:

```
  // 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_ );
```

7. 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.

```
        // 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.

        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
        // EBP to the caller's value.  Then jump
        // 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. You 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.

```
unit patterns;
#include( "pat.hhf" );



/************************************************************/
/*                                                          */
/* OneOrMoreCset-                                           */
/*                                                          */
/* Matches one or more characters in a string from          */
/* the specified character set.                             */
/*                                                          */
/* Disposition: Eager                                       */
/* BackTrackable:   Yes                                     */
/*                                                          */
/* Entry Parameters:                                        */
/*                                                          */
/*  ESI:    Pointer to sequence of characters to match.     */
/*  EDI:    Pointer to byte beyond last char to match.      */
/*  cst:    Character set to match with.                    */
/*                                                          */
/* Exit Parameters (if success):                            */
/*                                                          */
/*  EBX:    Points at the start of matched sequence.        */
/*  ESI:    Points at first character not in cst.           */
/*  EDI:    Unchanged from entry value.                     */
/*                                                          */
/* Exit Parameters (if failure):                            */
/*                                                          */
/*  EDI:    Unchanged from entry value.                     */
/*                                                          */
/* Unless noted, assume all other registers can be modified */
/* by this code.                                            */
/*                                                          */
/************************************************************/


procedure pat.oneOrMoreCset( cst:cset ); @nodisplay;
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.

begin oneOrMoreCset;

    // If some routine after this one fails and transfers
    // control via backtracking to this code, the stack
```

```
// will be a mess.  So save esp so we can clean up
// the stack if backtracking is necessary.

mov( esp, espSave );

// Save the pointer to the start of the string
// to match.  This is used as a "fence" value
// to prevent backtracking past the start of
// the string if things go really wrong.

mov( esi, startPosn );
mov( esi, ebx );

// Save pointer to end of string to match.
// This is needed to restore this value when
// backtracking occurs.

mov( edi, endStr );

// Okay, eagerly match as many characters in
// the character set as possible.

xor( eax, eax );
dec( esi );
repeat

    inc( esi );                 // Move to next char in string.
    breakif( esi >= edi );      // Stop at end of string.
    mov( [esi], al );           // Get the char to test.
    bt( eax, (type dword cst)); // See if in cst.

until( @nc );   // Carry is set if al in cst.

// So we can easily back track, save a pointer
// to the first non-matching character.

mov( esi, cursor );


// If we matched at least one character, then
// succeed by jumping to the return address, without
// cleaning up the stack (we need to leave our
// activation record laying around in the event
// backtracking is necessary).

if( esi > ebx ) then

    pat._success_( FailToSave, FailToHere );

endif;

// If we get down here, we didn't match at
// least one character.  So transfer control
// to the previous routine that supported
// backtracking.

mov( startPosn, esi );
pat._fail_( pat.FailTo );
```

```
        // If someone after us fails and invokes
        // backtracking, control is transfered to
        // this point.  First, we need to restore
        // ESP to clean up the junk on the stack.
        // Then we back up one character, failing
        // if we move beyond the beginning of the
        // string.  If we don't fail, we jump to
        // the code following the call to this
        // routine (having backtracked one character).

        FailToHere:

            mov( espSave, esp );    // Clean up stack.

            mov( cursor, esi );     // Get last posn we matched.
            dec( esi );         // Back up to prev matched char.
            mov( endStr, edi );
            mov( startPosn, ebx );
            if( esi <= ebx ) then

                // We've backed up to the beginning of
                // the string.  So we won't be able to
                // match at least one character.

                mov( ebx, esi );
                mov( FailToSave.ebpSave, pat.FailTo.ebpSave );
                mov( FailToSave.jmpAdrs, pat.FailTo.jmpAdrs );
                pat._fail_( pat.FailTo );

            endif;

            // If we drop down here, there is at least one
            // character left in the string that we've
            // matched, so call the next matching routine
            // (by jumping to the return address) to continue
            // the pattern match.

            mov( esi, cursor );
            mov( [ebp+4], eax );
            mov( [ebp], ebp );
            jmp( eax );


    end oneOrMoreCset;


    end patterns;
```

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.

```
    unit patterns;
    #include( "pat.hhf" );



    /*****************************************************************/
    /*                                                               */
    /* l_OneOrMoreCset-                                              */
```

```
/*                                                              */
/* Matches one or more characters in a string from              */
/* the specified character set.  Matches the shortest           */
/* possible string that yields (overall) success.               */
/*                                                              */
/* Disposition:     Deferred                                    */
/* BackTrackable:   Yes                                         */
/*                                                              */
/* Entry Parameters:                                            */
/*                                                              */
/*  ESI:    Pointer to sequence of characters to match.         */
/*  EDI:    Pointer to byte beyond last char to match.          */
/*  cst:    Character set to match with.                        */
/*                                                              */
/* Exit Parameters (if success):                                */
/*                                                              */
/*  ESI:    Points at first character not in cst.               */
/*  EDI:    Unchanged from entry value.                         */
/*                                                              */
/* Exit Parameters (if failure):                                */
/*                                                              */
/*  EDI:    Unchanged from entry value.                         */
/*  ESI:    Unchanged from entry value.                         */
/*                                                              */
/* Unless noted, assume all other registers can be modified     */
/* by this code.                                                */
/*                                                              */
/****************************************************************/



procedure pat.l_OneOrMoreCset( cst:cset ); @nodisplay;
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.

begin l_OneOrMoreCset;

    // If some routine after this one fails and transfers
    // control via backtracking to this code, the stack
    // will be a mess.  So save esp so we can clean up
    // the stack if backtracking is necessary.

    mov( esp, espSave );

    // Save the pointer to the start of the string
    // to match.  This is used as a "fence" value
    // to prevent backtracking past the start of
    // the string if things go really wrong.

    mov( esi, startPosn );
    mov( esi, ebx );

    // Save pointer to end of string to match.
    // This is needed to restore this value when
    // backtracking occurs.  If we're already
    // beyond the end of the chars to test, then
    // fail right away.
```

```
mov( edi, endStr );
if( esi >= edi ) then

    pat._fail_( pat.FailTo );

endif;

// Okay, this is a deferred version.  So match as
// few characters as possible.  For this routine,
// that means match exactly one character.

xor( eax, eax );
mov( [esi], al );           // Get the char to test.
bt( eax, (type dword cst)); // See if in cst.
if( @nc ) then

    pat._fail_( pat.FailTo );

endif;

// So we can easily back track, save a pointer
// to the next character.

inc( esi );
mov( esi, cursor );

// Save existing FailTo address and
// point FailTo at our back tracking code,
// then transfer control to the success
// address (jump to our return address).

pat._success_( FailToSave, FailToHere );




// If someone after us fails and invokes
// backtracking, control is transfered to
// this point.  First, we need to restore
// ESP to clean up the junk on the stack.
// Then we need to advance one character
// and see if the next char would match.

FailToHere:

    mov( espSave, esp );        // Clean up stack.

    mov( cursor, esi );         // Get last posn we matched.
    mov( endStr, edi );         // Restore to original value.

    // If we've exceeded the maximum limit on the string,
    // or the character is not in cst, then fail.

    xor( eax, eax );
    if
    {
        cmp( esi, edi );
        jae true;
        mov( [esi], al );
        bt( eax, (type dword cst ));
```

```
            jc false;
        }

            // 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 );

        endif;


        // If we drop down here, there is at least one
        // character left in the string that we've
        // matched, so call the next matching routine
        // (by jumping to the return address) to continue
        // the pattern match.

        mov( startPosn, ebx );
        inc( esi );                     // Advanced to next posn
        mov( esi, cursor );             // save for backtracking,
        mov( [ebp+4], eax );            // and call next routine.
        mov( [ebp], ebp );
        jmp( eax );

end l_OneOrMoreCset;

end patterns;
```