

Chris Lattner

Final Project: The HDLE Library

CS492 – Senior Seminar

Due: May 2nd, 2000

<u>Introduction to the Haskell Dynamic Lexer Engine.....</u>	2
Project Goals	2
<u>User/Programmer Interface Summary.....</u>	2
The DFA Module	2
DFA Constants.....	2
numNodes :: Int	2
emptyFA :: FA	2
DFA Data Structures.....	2
type FANode = (Int, [Int]) -- Not Exported	2
newtype FA = F [FANode] -- Ctor not exported.....	3
DFA Functions.....	3
matchDFA :: FA -> String -> Bool.....	3
matchDFAHead :: FA -> String -> String	3
matchDFALength :: FA -> String -> Int	3
matchDFALengthState :: FA -> String -> (Int, Int)...	3
showFA :: FA -> String.....	3
displayFA :: FA -> IO ()	3
measureFA :: FA -> IO ()	3
dumpFA :: FilePath -> FA -> IO ()	3
getFATransitions :: FA -> Int -> [Int]	3
getFAStateFinality :: FA -> Int -> Int	3
getNumStates :: FA -> Int	3
setFAStateFinality :: FA -> Int -> Int -> FA.....	3
setFA_ID :: Int -> FA -> FA	3
addNode :: FANode -> FA -> FA	4
nukeNode :: FA -> Int -> FA	4
appendFA :: FA -> FA -> FA	4
addFATransition :: FA -> Int -> Int -> Int -> FA	4
addFATransitions :: FA -> [(Int, Int, Int)] -> FA	4
regularNFA :: FA -> FA	4
completeDFA :: FA -> FA	4
removeDeadStates :: FA -> FA	4
The NFA Module	4
NFA Functions.....	4
addNFALTrans :: FA -> [(Int,Int)] -> FA	4
buildDFA :: FA -> FA	4
The RegEx Module	4
RegEx Functions.....	5
buildNFA :: String -> FA	5
compileRegEx :: String -> FA	5
measureRegEx :: String -> IO ()	5
matchRegEx :: String -> String -> Bool	5
The Lexer Module	5
Lexer Data Structures	5
type Token a = (String, a)	5
newtype Lexer a = Lexr (FA, [a]) -- Ctor not exported6	6
Lexer Functions.....	6
compileLexer :: [Token a] -> a -> Lexer a	6
lexFirstToken :: Lexer a -> String -> Token	6
lexIntoList :: Lexer a -> String -> [Token]	6
lexFile :: Eq a => Lexer a -> String -> IO [Token a] 6	6
measureLexer :: Lexer a -> IO ()	6
displayLexer :: Lexer a -> IO ()	6
serializeLexer :: Show a => Lexer a -> String -> IO ()	6
.....	6
<u>Selected HDLE Internals.....</u>	6
DFA Module Internals	7
NFA Module Internals.....	7
Internal NFA Functions.....	8
whereCanIGetFromHere :: FA -> [Int] -> Int -> [Int] . 8	8
getReachableTrans :: FA -> [Int] -> [(Int, Int)]	8
findBDFANode :: BDFA -> [Int] -> Int	8
RegEx Module Internals	8
Internal RegEx Data Structures	8
data TokType	8
Parsing Regular Expressions and Building NFAs.....	9
getFirstToken :: String -> String	9
tokenizeRE :: String -> [(TokType,String)]	9
buildTokNFA :: [(TokType,String)] -> FA	9
Lexer Module Internals.....	9
<u>Significant HDLE Examples.....</u>	10
RegEx Module Examples	10
Precompiled Regular Expressions	10
Dynamically Generated Regular Expressions.....	11
The Escape Mechanism	11
Lexer Module Examples	11
Processing User Input.....	11
<u>Conclusion.....</u>	13

Introduction to the Haskell Dynamic Lexer Engine

The Haskell Dynamic Lexer Engine (HDLE) was designed to help with Haskell's general text processing weakness. It provides a wealth of functionality in the area of text processing and token recognition, allowing powerful string parsing functionality to be used within the framework of a Haskell program. The HDLE also exposes lower level functionality for those applications that desire it.

This document assumes knowledge of topics in theory of computation, including Discrete Finite Automata (DFAs), Non-Determinate Finite Automata (NFAs), and Regular Expressions. User programmers do not need to understand the theory behind this library however: the theory just makes the internals of the library more apparent.

Project Goals

The HDLE is designed to be a powerful solution to Haskell's text processing weakness, while also providing superb performance. It achieves linear time text scanning, and nearly constant time scanner loading. It allows a Haskell programmer to dynamically compile and modify a lexer at runtime, allowing powerful operations. The project provides access to all levels of the finite automata generated, including the DFA, NFA, as well as the RegEx and Lexer composites.

The most important goals of the library are ease of use and performance.

User/Programmer Interface Summary

As hinted at above, the HDLE is composed of four different modules, and each provides a layer that builds on the others: their relationship is shown in Figure #1. Since each module may be used independently of the modules above it in the diagram, it makes sense to begin describing at the bottom and work up from there.



Figure 1

The DFA Module

The `DFA` module provides access to the Discrete Finite Automata engine used by the HDLE library. It exposes an Abstract Data Type, the `FA`, functions to operate on this ADT, and two exported constants that controls the `FA` generation techniques. The data structures used by the `DFA` module are fairly abstract, because the `NFA` module uses the same data structures, extending them by relaxing some of their restrictions.

DFA Constants

```
numNodes :: Int
```

This constant indicates the size of the `FA` language. Typically, the `DFA` module is configured to use a language consisting of all of the characters allowed to Haskell, characters `'\0'` through `'\255'` (which is set with a value of 256). At times, however, it is useful to limit the size of the `FA` to conserve memory and time resources. This is useful, for example when one knows that the only valid tokens accepted are in the ASCII range of `'\0'` to `'\127'` (which is configured with a value of 128).

```
emptyFA :: FA
```

This constant contains a definition for an "empty" finite automaton. This `FA` contains a single non-final, non-escaping node, and may serve as the basis for other more complicated `FAs` constructed with the member functions defined below.

DFA Data Structures

```
type FANode = (Int, [Int])    -- Not Exported
```

This structure contains the data needed for a single node in the FA. This information includes a finality number (the first value), and a list of transition values. The finality number contains information that indicates whether a state is final or not. (-1) indicates that the state is not final, any other value indicates the state is final. The exact value of the finality number may be returned by the match function, thus allowing one to tell *which* FA node matched the input string.

The list of transitions contains exactly numNodes entries for a valid DFA. Each entry of the list corresponds to a transition on the input value equal to its position in the list. For example, the first entry corresponds to the transition used for an input of the '\0' character. If an entry is undefined, because no transition exists, a value of (-1) is recorded. These values are treated as transitions to a non-escaping, non-final node in the graph.

```
newtype FA = F [FANode] -- Ctor not exported
```

The exported FA ADT is simply a list of nodes, with the non-exported “F” constructor. Each node transition in the FANode structure is defined as equal to the numerically defined node in the list of nodes.

DFA Functions

```
matchDFA :: FA -> String -> Bool
matchDFAHead :: FA -> String -> String
matchDFALength :: FA -> String -> Int
matchDFALengthState :: FA -> String -> (Int, Int)
```

This family of functions provides matching capabilities for a DFA, and are the most likely to be used. matchDFA simply returns a Boolean indicating whether or not a string matches a DFA. matchDFAHead returns the longest matching string that comes from the beginning of the string parameter. This is useful for matching tokens off the head of a string, looking for the longest match. matchDFALength is similar to matchDFAHead, but returns the length of the match, instead of the string itself. matchDFALengthState returns two integers, the first is the length of the match, and the second is the finality number of the matched state.

```
showFA :: FA -> String
displayFA :: FA -> IO ()
measureFA :: FA -> IO ()
```

The showFA function turns the very unwieldy FA structure into a nice and human readable format. This function implements “show” for the ADT. displayFA simply calls putStr on the result of showFA to evaluate the embedded newlines for the programmer. measureFA measures the FA and returns statistics, including the number of states, transitions and final states.

```
dumpFA :: FilePath -> FA -> IO ()
```

dumpFA writes an FA out to a file in a format that may be turned into a module and imported. This allows the programmer to create precompiled FAs.

```
getFATransitions :: FA -> Int -> [Int]
getFAStateFinality :: FA -> Int -> Int
getNumStates :: FA -> Int
```

getFATransitions returns the list of transitions from a specified node in the FA. getFAStateFinality returns the finality number of a particular state. getNumStates returns the number of states in an FA.

```
setFAStateFinality :: FA -> Int -> Int -> FA
setFA_ID :: Int -> FA -> FA
```

setFAStateFinality sets the finality number of a specified state. Its parameters are: source FA, state, and finality number. setFA_ID is an interesting function that changes the finality numbers of all finality=0 states to the specified finality number. Again, it **only** effects states with a finality number of 0.

```

addNode :: FANode -> FA -> FA
nukeNode :: FA -> Int -> FA
appendFA :: FA -> FA -> FA

```

This group provides simple functions for modifying nodes in a FA. `addNode` appends a node to the end of an FA, `appendFA` sticks two FAs together, and `nukeNode` deletes a node out of an FA. The `nukeNode` and `appendFA` functions adjust the transition numbers of the remaining nodes in the resultant FA so that they still refer to the correct nodes in the source FAs.

```

addFATransition :: FA -> Int -> Int -> Int -> FA
addFATransitions :: FA -> [(Int, Int, Int)] -> FA

```

`addFATransition` adds a single transition to an FA. Its parameters are: input FA, from node, to node, and transition value. `addFATransitions` adds a list of transitions to the FA in one operation, each entry in the list contains a from node, to node, and transition value.

```

regularNFA :: FA -> FA
completeDFA :: FA -> FA
removeDeadStates :: FA -> FA

```

`regularFA` transforms an NFA such that the only final state is the last one, and all final states have a finality of 0. `completeDFA` “completes” a DFA by making sure that no transitions with a value of (-1) exist in the FA (by inserting a new dead state at the end). `removeDeadStates` performs the inverse of `completeDFA`, by removing all nodes that are non-final and not escapable, and setting any transitions into them to (-1).

The NFA Module

The NFA module builds on the powerful DFA module to provide Non-deterministic Finite Automata to the HDLE library. NFAs are identical to DFAs, with the exception that suddenly each state may have more than `numNodes` transitions in them, each additional transition is considered to be a *lambda* transition, and as such, an unbounded number may be present in each state. Because NFAs are a strict superset of DFAs, many of the functions exported by the DFA module are also appropriate to be used on NFAs (these are indicated by saying they work on FAs instead of strictly on DFAs).

NFA Functions

```
addNFALTrans :: FA -> [(Int,Int)] -> FA
```

This function is used to add a set of lambda transitions to an NFA [The parameter is a list of (from, to) pairs]. Note that the DFA functions `addFATransition(s)` also allow the addition of lambda transitions by specifying a transition value of (-1).

```
buildDFA :: FA -> FA
```

This is the main workhorse of the NFA module. It allows one to convert from an NFA representation a strict DFA representation of the language.

The RegEx Module

The RegEx module consists of a regular expression parser and NFA data structure generator. The regular expression syntax recognized is a powerful one that is useful and simple in syntax. Below is a summary of the recognized syntax, starting with the primitive values.

Syntax:	Function:
'.'	Match any character
"[...]"	Match any of the characters listed in the brackets
"[^...]"	Match any character, except those listed in the brackets

“x”	Match a literal character.
-----	----------------------------

Many operators are recognized by the regular expression grammar. The two infix operators take two parameters and are left associative; the postfix operators take a single parameter and are right associative.

Precedence:	Operator:	Function:
Highest	‘()’ – Parentheses	Grouping, altering other precedence rules.
High	‘ ’ – Alternation [infix]	Match either one of its two parameters
High	‘-’ – Subtraction [infix]	Match everything in the language to the left, except for strings in the language to the right. Ex: “. -a” is anything but ‘a’.
Medium	‘*’ – Klein Enclosure	Match zero or more of its parameter
Medium	‘+’ – Repetition	Match one or more of its parameter
Medium	‘?’ – Option	Match zero or one of its parameter
Medium	‘!’ – Inversion	Match anything but its parameter
Low	“xy” – Juxtaposition	Match ‘x’ followed by ‘y’

The `RegEx` module only exports one interesting function, the rest are simply convenience functions to allow programmers to work at a higher level than the `DFA` or `NFA` modules (it is expected that most programmers will work at either the `RegEx` or `Lexer` level of the HDLE library).

RegEx Functions

```
buildNFA :: String -> FA
```

`buildNFA` takes a string regular expression as input and returns a constructed NFA as output. This is the only “interesting” function of the module.

```
compileRegEx :: String -> FA
```

```
measureRegEx :: String -> IO ()
```

```
matchRegEx :: String -> String -> Bool
```

These are the convenience functions provided by the module. `compileRegEx` compiles the regular expression down to a DFA by calling `buildDFA` after `buildNFA`. The `measureRegEx` function compiles a regular expression down to a DFA, then measures the result and `putStrs` it. The `matchRegEx` function compiles a regular expression (the first argument), and then matches against the second argument, returning the result. This function is good for quick and dirty matching when performance is not an issue (if performance is an issue, then the host application should cache the compiled DFA).

The Lexer Module

The `Lexer` module builds on all those below it to provide a clean and simple package for common lexical analysis tasks. It allows a `Lexer` to be built on the fly, compiled and written to disk, or loaded from disk and made ready to run. The lex functions that it provides are simple and convenient to use, allowing the most power out of the fewest lines of code. The resulting code stream is simple to parse and quick to consume.

A `Lexer` is built by calling the `compileLexer` function with a list of pairs and an error token. The pairs consist of a regular expression and a token to return for each regular expression when it matches. The token can be an arbitrary user defined type. The error token is returned if no regular expression matches. See the Examples section at the end of this document for an example of use.

Lexer Data Structures

```
type Token a = (String, a)
```

This is a very important type for the `Lexer` module. The list of pairs passed to the `compileLexer` function is composed of `Token` elements. When a string is lexed, values are returned as `Tokens`. Each token consists of a string, and the user defined value that corresponds to it.

```
newtype Lexer a = Lexr (FA, [a]) -- Ctor not exported
```

The `Lexer` type consists of two things: a DFA and a mapping of finality numbers to user defined types. The first `Lexer` value is a composite DFA that corresponds to all of the regular expressions, each set with a different finality number. The second `Lexer` value is a mapping from the sequentially assigned finality numbers back to the user-defined type. This allows the user to use their UDTs natively without having to bother the underlying layers with them. The first entry of the list is the error token, which is returned if no match is possible.

Lexer Functions

```
compileLexer :: [Token a] -> a -> Lexer a
```

This is the workhorse for the `Lexer` module. It compiles all of the input regular expressions into a DFA and sets of the user-defined type-mapping table.

```
lexFirstToken :: Lexer a -> String -> Token
lexIntoList   :: Lexer a -> String -> [Token]
lexFile       :: Eq a => Lexer a -> String -> IO [Token a]
```

These functions lex a set of input characters and return either a token or a list of tokens. `lexFirstToken`, lexes and returns the first token from the string. It returns the error token at the end of the string. `lexIntoList` lexes the entire string, returning each token as part of a list. This is more useful for two reasons: 1. Haskell's lazy evaluation makes this work very efficiently (constant space overhead) and 2. The end of the stream is determined by the end of the list, leaving the error token to indicate actual errors. `lexFile` essentially reads a file and hands it to `lexIntoList`.

```
measureLexer :: Lexer a -> IO ()
displayLexer :: Lexer a -> IO ()
```

`measureLexer` measures the size of the DFA and prints the information to standard output. `displayLexer` displays the contents of the lexer in a human readable (debugable) form.

```
serializeLexer :: Show a => Lexer a -> String -> IO ()
```

`serializeLexer` serializes a lexer to a file, for later reuse. This is extremely useful because Lexers tend to take a long time to compile, and all of that information is static for most programs. The input string is the name of the file and the given name of the lexer that is later imported. Later, the program may reuse the lexer by simply importing the module into its program. There is one caveat, however.

If the program uses the lexer with a user defined type, such as a data declaration, `serializeLexer` cannot add its declaration to the output file. The solution to this is to manually edit the resulting file to import whatever module contains the definition of the dependent UDT. Also, the UDTs must show to a readable form.

Selected HDLE Internals

This section of the document is meant to describe the inner workings of the HDLE library. It is not meant to exhaustively document the library (that is what comments are for), nor is it meant to make the reader an expert in regular expressions and lexers. It is meant purely to give a taste for the implementation details that were faced and the intermediate data structures used for each process.

DFA Module Internals

Given a description of the FA data structure, most of the functions are pretty short and straightforward. The most interesting function is `matchDFALengthState`, and its helper function `matchDFAh`, shown below:

```
-- matchDFALengthState - Match the specified DFA against the head of the
-- specified string. This returns the length and the DFA ID that the
-- string ended up in. If the match fails, then the DFA ID = -1.
--
matchDFALengthState :: FA -> String -> (Int, Int)
matchDFALengthState (F dfa) = matchDFAh (dfa!!0) dfa

-- matchDFAh - Helper function for matchDFALength. This function does all
-- the hard work of seeing if there is a transition from the current state
-- matching the current character. If there is, we continue matching at the
-- transition destination state. If not, we check to see if this state is
-- final. If it is... then we return (0, DFA_Id) to signify we match an
-- empty string. If it is not, then we return (0,-1) to indicate we did not
-- match at all.
--
matchDFAh :: FANode -> [FANode] -> String -> (Int, Int)
matchDFAh (final, _) _ "" = (0, final)      -- Empty string...
matchDFAh (final, transitions) dfa (s:str)
  | blocked && (final /= (-1)) = (0, final)
  | blocked   = (1+length str, -1)          -- Return the remaining string as error
  | otherwise = (sLen+1, sFin)
  where
    -- trans is the transition for the 's' character
    trans = transitions!!(ord s)
    (sLen, sFin) = matchDFAh (dfa!!trans) dfa str
    blocked = trans == (-1) || sFin == (-1)
```

The `matchDFALengthState` function (and indeed, all of the match functions) is a simple wrapper function around the helper function. The helper function does the hard work of finding the longest matching substring of the input that matches the DFA.

`matchDFAh` takes an `FANode` parameter, which is the current node, a list of `FANodes`, which is the FA without a wrapper, and a string to match against. It returns a two-tuple that contains a match length and a finality number of the state that is returned. It works by running through the DFA, keeping track of the last final node that is traversed through before making an illegal transition. As such it contains two cases: an end of string base case, and a recursive case.

The base case simply returns a length of zero, and a finality of the current state. If the current state is final, then it returns a final finality state, if not, it returns `(-1)`. The recursive case for this function is slightly more complicated. First it checks to see if the transition is blocked: Either not having a transition out of the current state on the input, or by the rest of the input not matching. If the rest of the input string matches, then a tuple containing the submatch final state and the incremented length of the matched string is returned.

Note that if a DFA is “complete” (i.e. it has no undefined transitions), then the entire string must be read to detect a match termination condition (which is inefficient). Because of this, all NFAs have their “dead” states removed by the `removeDeadStates` function before they are returned.

NFA Module Internals

The NFA Module provides one interesting function, the `buildDFA` function. The code is too involved and complex to include here, but it uses a well-known algorithm that keeps track of the different paths that can be used to get to each node. In the process, it builds a DFA that is supplemented with additional

information: it must keep track of when a node has been visited through the same NFA path more than once. To support this, it uses an internal data structure, the `B DFA` and `B DFANode`:

```
type BDFANode = (Int, [Int], [Int])
type B DFA = [BDFANode]
```

These two data structures are used to serve the *exact* same purpose as the `FANode` and `FA` data structures, except that they are used only internally, and that the node contains an extra data item. This value is a list of integers that tell the builder which NFA nodes the current DFA node corresponds to. Thus, when the algorithm is tracing through all possible routes of the NFA, if a cycle is encountered (highly likely), the single node that corresponds to the cycle path is recycled and used again. This prevents the DFA from growing to an unbounded size. Once the `B DFA` has been built, the third data member is stripped off and the result is returned as a normal DFA.

Internal NFA Functions

```
whereCanIGetFromHere :: FA -> [Int] -> Int -> [Int]
```

This function looks at an NFA and evaluates which states may be reached from the current state (third parameter) by only taking lambda transitions. The second parameter is a list of states already visited and is supplied to this function as `[]`. The list of already visited states is kept track of to avoid infinite recursion on an NFA lambda loop.

```
getReachableTrans :: FA -> [Int] -> [(Int, Int)]
```

This function returns a list of all of the transitions that may be taken from a list of states in the NFA. This list of states is generated by the `whereCanIGetFromHere` function, and the list of transitions corresponds to all of the possible places we could get to if we read a character from one of these states. The list returned is a list of pairs, where the first value is the transition value, and the second is the transition destination.

```
findBDFANode :: B DFA -> [Int] -> Int
```

`findBDFANode` searches through a `B DFA` to see if it can find a state that has already been visited. If not, a value of `(-1)` is returned. It detects a match by comparing the signature list of NFA states with each node's saved list.

RegEx Module Internals

The `RegEx` module is the most complicated module, at least based on bytes of code. In addition to containing several internal functions, the `RegEx` module actually uses a DFA to parse the regular expressions! Because of space and time limitations, I will only describe a few of the `RegEx` module's pearls (for more, read the code).

The `buildNFA` function basically boils down to a composition of two interesting functions: the `tokenizeRE` function and the `buildTokNFA` function.

Internal RegEx Data Structures

```
data TokType
  = TokChar | TokDot | TokCharClass | TokAlternation | TokSubtract |
    TokStar | TokPlus | TokOption | TokInvert | TokConcat | TokRegEx
```

These values correspond to tokens returned by the `tokenizeRE` internal parser function. The first three values correspond to primitives, the second two are infix operations, and the rest are binary operations. The `TokRegEx` value is returned when a regular expression type has yet to be defined, for example when parentheses are used or when parsing other expressions. As an example, the expression `"a|b"` would be parsed into `[TokRegEx ("a"), TokAlternation, TokRegEx ("b")]`, and the concrete types are determined at a recursive parse stage.

Parsing Regular Expressions and Building NFAs

```
getFirstToken :: String -> String
```

The `getFirstToken` function provides essentially the same function as the `lexFirstFunction` does for lexers. Basically, it reads the first token out of the input text stream and returns it. The tokens may be a single character, a recursively parenthesizable subexpression, or a character class. Ideally it would be nice to use a DFA to match all three cases, but unfortunately, a recursive parenthesis matcher is not a regular language, so it cannot be represented with a DFA. Instead, the parenthesis case is a simple little function, the char class case uses a DFA, and the single character is the base case.

The DFA used to match the bracket token (`brackDFA`) is relatively complex, because it must consider escaped characters, and must take the first unescaped `']` that it finds as the end of the char class... it may not glob until it finds the last `']` character in the input stream. As such, it is equivalent to the `"([^\]]\\|\\.)*"` regular expression.

```
tokenizerRE :: String -> [(TokenType,String)]
```

The `tokenizerRE` function uses a simple recursive parser to convert the input string into a precedence respecting parse tree. It takes the regular expression and tokenizes it into pairs of `TokenType`s and the actual text that it matches, quite similar to the lexer. Although a text representation is included for the operators, it is not needed, as the token type determines the text that matches it. An example of this use is: `tokenizerRE "cat|dog" = [(TokRegEx,"cat"), (TokAlternation,"|"), (TokRegEx,"dog")]`.

This function works by scanning first for the infix tokens, then for the postfix tokens (to preserve precedence relationships). Eventually, each regular expression boils down to one of the primitive base cases: `TokChar`, `TokDot`, or `TokCharClass`.

`TokChar` and `TokDots` are easy to process, but `TokCharClasses` are not quite as straightforward.

```
buildTokNFA :: [(TokenType,String)] -> FA
```

The `buildTokNFA` function takes the tokenized regular expression string and returns an NFA that corresponds to that result. This NFA has a constraint on it that the NFA must start at the first state and end at the last state of the FA, facilitating recursive NFA builds. The `buildTokNFA` function recursively calls the `buildNFA` function, making the `buildNFA` and `buildTokNFA` functions mutually recursive.

The `buildTokNFA` function is implemented using Haskell pattern matching to match different operators. For example, the parse returned for the alternation operator is shown above. The pattern match for this appears as follows:

```
buildTokNFA [(TokRegEx, s1), (TokAlternation,_), (TokRegEx, s2)] = ...
```

This elegant solution makes the code very modular and easy to follow as well as allowing easy extensions of the regular extension grammar. All that must be done is to add an extra rule to the `tokenizerRE` function and add an extra match to the `buildTokNFA` function.

Lexer Module Internals

Like the other modules, the `Lexer` module only has one interesting function, and the rest are pretty simple glue functions. This function is the `compileLexer` function. The `compileLexer` function takes as input a list of `RegEx/Token` pairs and creates a composite DFA that represents all of the `RegEx` DFAs. Complete source for it and its helper function is included below:

```
compileLexer :: [Token a] -> a -> Lexer a
```

```

compileLexer toks err =
  Lexr (mergeFAs (zipWith setFA_ID [0..]
    (map (buildNFA.fst) toks)), err:map snd toks)

mergeFAs :: [FA] -> FA
mergeFAs = buildDFA . foldl addFA emptyFA
  where
    addFA :: FA -> FA -> FA      -- Add an FA to the meta FA
    addFA oldFA newFA =
      addFATransition (appendFA oldFA newFA) 0 (getNumStates oldFA) (-1)

```

As you can see, `compileLexer` starts by first compiling each of the regex's into their corresponding NFA form. Then it sets the finality number of each NFA to be an increasing digit, and finally it merges them all together.

The `mergeFA` function performs the task of putting each discrete NFA together into a useful whole. It does this by creating a new empty state at the start of the NFA, then appending each NFA onto the end of it, adding lambda transitions to the start of each regex NFA as it goes. Once all of the regular expressions are built into a single NFA, the NFA is compiled into a DFA and returned.

Significant HDLE Examples

Because this is complex information, the only realistic way to learn how to use it is through examples. This section proposes and examines a few concrete examples to demonstrate how the functions are used and how they relate. Because the `DFA` and `NFA` modules are rarely used directly (and because the `NFA` and `RegEx` modules are good examples of their usage), examples for them are excluded from this section.

RegEx Module Examples

The `RegEx` module provides simple functions for creating and manipulating compiled regular expressions. There are two very common patterns of usage for regular expressions in applications today: constant predetermined regular expressions and dynamically generated regular expression patterns. Examples for the `RegEx` module will concentrate on these two usage cases.

Precompiled Regular Expressions

Precompiled regular expressions are typically used in applications that are parsing a fixed format data file, and must rapidly assimilate the information (for example, a log file analyzer). In this case, the regular expressions are known at application compile time and are fixed: the user cannot modify them with settings or other user input. Because usage of regular expressions is so common, the HDLE Library provides a way of compiling a regular expression, then serializing it out to a file. The following code snippet does just this:

```

brackFA = compileRegEx "([^\]\\\\\\\\]|(\\.\\.))*]"
dumpFA "brack" brackFA

```

This code creates a module named `"brack.fa"` and writes the low-level definition of the underlying finite automata to it. Later, if you want to use the serialized FA, use code that looks like this:

```

import "brack.fa"
matchDFA brack "\\]\\\\-]" -- Returns True

```

Note that the `"import"` process is much faster than the original compilation process, especially for complex regular expressions.

Dynamically Generated Regular Expressions

The other common class of regular expression utilization is dynamically generated regular expressions. Two examples of this are a `grep` like utility, where the user inputs `Regex`'s directly, and a scripting language with intrinsic support for regular expressions. In the later case, the `Regex`'s are input as tokens in the input script – these tokens are parsed at runtime and compiled into regular expressions.

With this class of usage, there is little opportunity to cache precompiled regular expressions. The only possible way to improve performance in this situation, then, is to intelligently reuse the compiled form internal to the application: For example, if the script is running a loop with a regular expression in it, only compile the `Regex` once, and match multiple times against the compiled form.

The Escape Mechanism

Several characters are reserved as operators in the regular expressions, including `'+'`, `'-'`, `'*'`... etc. Because of this, there must be an alternative way to represent a match of these literal characters. To do this, the character class mechanism supports escape sequences, and also has fewer operators defined in general. Here are some examples:

Pattern:	Matches:
<code>compileRegex "+"</code>	<code>"() +"</code> - Zero or more empty strings (i.e. an empty string)
<code>compileRegex "[+]"</code>	The plus character
<code>compileRegex "-"</code>	<code>"() - ()"</code> – Nothing, no final states
<code>compileRegex "[-]"</code>	The minus character

As you can see, the character class mechanism provides a general matching function that can be used to work around reserved operators.

Lexer Module Examples

A lexer is a very useful tool that may be used for many more applications than just compiler design. Because Haskell is generally weak with text processing, this lexer can be used as a convenient way of processing command line input, which may be fraught with user input errors.

Processing User Input

Presented here is a simple program that accepts user input consisting of the following keywords: `load`, `help` and `quit`... each of which may be abbreviated with a single letter. The `load` command is followed by a filename to load. This program is not designed to do meaningful work, just as an example of processing text.

There are several things that complicate processing of user input. The first is the backspace character: because Haskell does not buffer IO, backspaces are sent through as raw input and must be stripped by the application. This is not an insurmountable problem, as it is taken care of with a single function.

The second is the parsing stage of the input. Although they are no substitute for a full-blown parser, Haskell's pattern matching rules are very powerful for handling user input.

The third issue is how to handle errors. This is taken care of with two different techniques: a general error handler and a specific error handler to produce user-friendly error messages. The general technique just uses a pattern matching rule that always matches, and prints the standard "unrecognized input" message. The context specific error handler is used to catch common input errors and gently assist the user.

A user transaction with the program is illustrated below, with user input bolded:

```
Main> go
Input (or help)>help
Supported commands: quit, help, load "filename"
```

```

Input (or help)>What do I do?
Invalid input: What do I do?
Input (or help)>l
The load command must be followed by a filename!
Input (or help)>load "filename"
Loading "filename" as you requested
Input (or help)>okay, quit me
Invalid input: okay, quit me
Input (or help)>quit
Main>

```

...and here is the source code for the program:

```

import Lexer

-- stripBS processes the backspace characters in a string...
stripBS (x:'\b':xs) = stripBS xs
stripBS (x:xs) = x:stripBS xs
stripBS [] = []

-- go, the main distributor program...
go = do putStr "Input (or help)>"
      input <- getLine
      result <- processInput (stripBS input)
      if (result) then
        return ()
      else
        go

-- Token types for the lexer..
data TokType = TokIgnore | TokHelp | TokQuit | TokError | TokLoad | TokString
              deriving (Show, Eq) -- Make it showable, and =='able

tokLexer =
  compileLexer [ ("[\n\t]++", TokIgnore), -- Ignore whitespace
                ("load?", TokLoad), -- Match load
                ("help?", TokHelp), -- Match help
                ("quit?", TokQuit), -- Match quit
                ("\"[^\"]*\"", TokString)
              ] TokError

-- processInput - Convert string into a list of tokens, remove ignored
-- tokens, then pattern match the result.
--
processInput = processTokens . filter ((/=TokIgnore).snd) . lexIntoList tokLexer

-- Pattern match the input tokens...
--
processTokens [(_, TokQuit)] = return True
processTokens [(_, TokHelp)] =
  do putStr ("Supported commands: quit, help, load \"filename\"\n")
  return False
processTokens [(_, TokLoad), (v, TokString)] =
  do putStr ("Loading " ++ v ++ " as you requested\n")
  return False
-- Match a specific, common error case...
processTokens [(_, TokLoad)] =
  do putStr ("The load command must be followed by a filename!\n")
  return False

-- Match for the general error case...
processTokens x = do (putStr . (++ "Invalid input: " . concat . map fst) x
                    putStr "\n")
                    return False

```

Conclusion

The Haskell Dynamic Lexer Engine provides a wealth of functionality to the Haskell language. It simplifies text processing a great deal and enables simplified parsing of expressions. It is designed to be simple, elegant and extensible, and hopefully other people will appreciate its utility.