
CS 421 – Spring 2007

Lecture Notes Set 18: Regular Expressions

Elsa L. Gunter¹

Transcribed by: Pooja Mathur

Corresponding to Slides: 16 - ocamllex (slides 16 - 23)

Made available: March 2, 2007

Revision 1.0

1 Change Log

1.0 Initial Release.

2 Syntax for Regular Expressions (slides 16 - 19)

Here we will talk about the syntax for making up regular expressions. We talked about a few last time, but here we will go into more detail.

So, if you see a character in single quotes, that is representing a single character. It looks like this:

'a'

Next is the underscore. The underscore can match against any character that is allowable in your language. If there was no underscore, then one would have to make a case for each and every character in the language and specify an action for each of them.

There is a character set, Eof that means you have reached the end of the string of characters. There are no more characters left to parse. This is different than the empty string character. It is a special character meaning that there are no more characters coming in after this.

You can place regular expressions in parentheses. If you want the Klene Star of an expression that is more than one character, the parentheses are needed to group the expression together and then apply the Klene Star. Otherwise, the Klene Star may only be applied to the character right before the star.

You can place two regular expressions next to each other and that will concatenate the expressions. So you can take a string that is described by one regular expression and a string described by some other regular expression and if you place the two strings side by side, then that is the string describe by the concatenation of the two expressions.

You can make a 'string' by simply concatenating a sequence of characters. That is, if you wanted to define the keyword, 'if', you don't have to write 'i'f'...etc. You can simply write the concatenation of the two characters in single quotes, 'if'.

Since the logical 'or', \vee , isn't technically an ASCII character, we define the 'or' of two characters to use the verticle bar, |, instead. So if you were trying to say $e_1 \vee e_2$, that would be written as $e_1 | e_2$. So here, we are saying that we can take a string that is described by the first regular expression, or the string that is described by the second regular expression.

Now, if you wanted to describe a choice in a collection of characters, you would use square brackets. And the first character and the last of the collection should be placed in the square brackets with a hyphen in between them. Also, the characters should be in single quotes. So if someone wanted to describe the first five letters of the alphabet, they could write [`'a'-'e'`]. This is the range of characters between *a* through *e*.

The complement of a set of characters, or range of characters is given by the \wedge symbol. So if you wanted all the characters that weren't *a* through *e*, you could write it as [`^'a'-'e'`]. Everything stays the same as before, the only

¹© 2007, Share and Enjoy

difference is that the carat symbol is in there. Or, if you wanted to describe the set of characters that not digits, that would be written as `[^'0'-'9']`.

The `*` stands for the Klene star. This says that character or set of characters before it can be repeated zero times or more. That is, you either get the empty string, or you get some number of repetitions of that character. So if you had `('a' | 'b')*`, that is, a or b , this could produce numerous different strings. You can get the empty string, a , b , ab , ba , etc. Note that everytime you swing around to choose again, you get to make a fresh choice. What ever characters are in that range are available each time.

The Klene plus basically takes out the option of the empty string. The first time around when looking at something in the Klene plus, you have to choose a character, there is no getting around it. Then after that, you can choose the empty string if you like, or any other character available.

The question mark is giving you an option for a character. It is saying that the character can be there, but it doesn't have to be. That is, you can choose the character in question, or you can choose the empty string. So, an example of this would be when writing floats. When we talked about floats last time, we say that a float was a sequence of digits, followed by a decimal point, and then another sequence of digits. With the question mark, you have the the first set of digits be optional. That is, instead of having to write 0.5 you can just write $.5$ and that will still make sense.

Now, let's take a look at the pound operator. If you have two sets of characters, e_1 and e_2 . We can say $e_1 \# e_2$ to mean that we want the characters that are in e_1 and not in e_2 . The operator can only be applied to regular expressions that describe sets of characters. Using this, let's try to describe the set of characters that are not alphanumeric. Well, to start off, let's describe the set of digits. So we'll write:

```
let digit = ['0'-'9']
```

Now we will write both sets of upper and lower case letters.

```
let lower = ['a'-'z']
```

```
let upper = ['A'-'Z']
```

Now we have our sets of characters. Using these, we can describe alphanumerics with:

```
let alphanum = digit | lower | upper
```

So, to describe the characters that are no alphanumeric:

```
let not_alphanumeric = _ # alphanumeric
```

This is saying, with the underscore, that we'll initially take any possible character and then throw away any upper case letter, lower case letter, or digit. So we end up with the set of characters that are not alphanumeric.

Next, you can use identifiers to make it easier to refer to regular expressions. So you can write, just like we were doing in the last example, `let identifier = regular expression`.

One more thing you can do is e_1 as identifier. What this is doing is similar to a `let ... = ... in ...`. This is a local binding of some regular expression to an identifier. This is local to the action that is will occur if this expression is seen.

Now, there are a few more things you can do with OCamllex. And those can be found online at the inria website.

3 Example

3.1 Code (slides 20 - 21)

So here is the code of an `.mll` file. Like we talked about last time, `.mll` files start with the header. One of the main things that we want in a file are type declarations. Especially important are those that are involved in the actions after finding regular expressions. So that is what we do first, we create the result type. This is going to be the type of the output of the actions..

```
{ type result = Int of int | Float of float | String of string }
```

You also might need auxiliary functions, but here we simply created identifiers for regular expressions. So, we define `digit`, as any one character from the numbers 0 through 9. Then there is `digits`, that is defined as one or more characters from the `digit` set. `Lower_case` are the set of characters from a through z , while `upper_case` is the set if characters from A through Z . `Letter` is described as either a character from the `lower_case` set or from the `upper_case` set. And then `letters` is one of more charaters from the letter set. Note that anything that starts with an upper case letter is allowable here, even though, regularly in OCaml, words that start with `upper_case` letters are reseeded words.

```

let digit = ['0'-'9']
let digits = digit +
let lower_case = ['a'-'z']
let upper_case = ['A'-'Z']
let letter = upper_case | lower_case
let letters = letter +

```

Now we have the parsing function. We begin the same as always, with the rule keyword, main, and then we say that we are going to start parsing. Then we say if we see digits, followed by a decimal point and then more digits, we have the action to return that as a float. Then we a rule for if we just see digits, we return that as an integer. And then finally, if we see letters, we return that as a string. There is also the rule to call main on the buffer if we see any other character than those described here.

Note that we choose the rule that satisfies the longest string we can get. If there are two rules, and the string that is satisfied is the same length for both rules, then the one that comes first is chosen.

Each action returns something of type result. For your lexer, all the actions have to return the same type. Here the type is the type we declared in the header.

```

rule main = parse
(digits)'.digits as f { Float (float_of_string f) }
| digits as n { Int (int_of_string n) }
| letters as s { String s }
| _ { main lexbuf }
{ let newlexbuf = (Lexing.from_channel stdin) in
print_string "Ready to lex.";
print_newline ();
main newlexbuf }

```

3.2 Output (slides 22 - 23)

So to use it , you say you and then the name of your lexer. It will do some stuff, and then ask you for the input buffer string. Our input is 'hi there 234 5.2' and our output is a String of 'hi'. But where did the rest of our output go, we should have gotten another string and then an integer and than a float.

```

# use "test.ml";;
...
val main : Lexing.lexbuf → result = <fun>
val _ocaml_lex_main_rec : Lexing.lexbuf → int → result = <fun>
Ready to lex.
hi there 234 5.2
- : result = String "hi"

```

Well, the key here is that we only used the buffer once. Our rule isn't a recursive rule. When we find a character to go into the underscore rule, we aren't exactly calling main lexbuff again, what we are stopping. We see a space and then we stop. And then the rest of our input string gets thrown away.

So, to fix this, we put our string into an identifier. The string is 'hi 673 there'. And then we call main on the identifier. And our first return value is the result String 'hi'. We saw the characters *h* and then *i* and then a space. We don't do anything with the space, as soon as we see the space, we know the character before this one is the end of the string we were looking for. Then, if we were to check our identifier, we would see that the 'hi' is gone now and all we have is '673 there'. So, now we can call main again and get the result of Int 673. And then finally we can call main one more time, and finish off the string and get back String 'there'.

```

# let b = Lexing.from_channel stdin;;
# main b;;
hi 673 there
- : result = String "hi"
# main b;;
- : result = Int 673

```

```
# main b;;  
- : result = String "there"
```

4 Problem (slides 24 - 25)

So, how can we change the lexer to repeat, without having the user have to call the lexer over and over again. The answer is in the action portion of lexing.

What we can do is use the end of file character for like a basic case for a recursive call. And then have the actions return each step as you move along.