ECE251: Programming Languages & Translators Fall 2010

Lecture 5 — September 22, 2010

Patrick Lam *version 2*

Back to text-processing domain-specific languages. Our examples:

1. web scraping and rewriting: replace URLs with links to print-friendly pages.

2. Cucumber testing framework: parse strings like "**When** I go to 'Accounts' **then** I should see link 'My Savings'."

3. Lexical analysis: [while] [(] [x] [<] [5] [)] [{] [x] [++] [;] [}]

4. Command-line manipulation: `${dir}/foo -> /home/plam/foo`

5. Search and replace (new): replace ′ with ' when surrounded by caps, e.g. `'ignore these′ O′Lam -> 'ignore these′ O'Lam`

We have seen the underlying engine, *regular expressions*. How will we use the engine? Where do the FA come in?

- *Accept*: match the entire string.

  Q: Does the *entire* string s match a pattern r?

- *Match*: match a prefix of a string.

  Q: Does some *prefix* of s match a pattern r?

- *Substring*: match some substring of a string.

  Q: Does some *substring* of s match a pattern r?

- *Tokenize*: split into matches of different expressions.

  Q: How can we express s as a *partition* of lexemes $r \in R$?

- *Extract*: like match and search, but extract the substring.

  Q: *Which part* of s matches matches a pattern or subpatterns r?

- *Replace*: like extract, but replace matched pattern.

  Q: *What is* the string s' which has the patterns r' replaced as specified?

Which of these questions do we need to answer to solve the above problems?

Example. Let's investigate example (5), search and replace. Goal: `‘ignore these’ O′Lam -> ‘ignore these’ O’Lam`

That is, match O′L, or actually any two `[A-Z]`s with a ′ in between. Python code:

```
import re;
text = "‘ignore these’ O′Lam";
print (re.sub(r"([A-Z])[′‘]([A-Z])", "\\1\x92\\2", text));
```

Another Example. What about lexing? What we are really doing:

Partition the input into a sequence of matches of the alternatives of the regexp.

Problem: partition may not be unique, e.g. `map -5 pi`. If we allow negative numbers, is `-` part of `-5` or is it the operator `-`?

Another problem case is C++ templaces, `List<List<int> >`.

A solution: ensure unique partition (get rid of ambiguity) by e.g. using *maximal* match. (Is this always what we want?)

Yet Another Example. Consider JavaScript.

```
[a-zA-Z][a-zA-Z]* | / | /[^/]*/g}
```

Is `...e/f/g...` the sequence ID, DIV, ID or ID, REGEX? Need input from the parser.

Regular expressions vs. regexps

So far we've seen "regular expressions" and their translation into NFAs, DFAs, etc. We've also seen how we can apply them to text processing.

But no one implements just regular expressions!

For instance, you can do primality testing (of unary numbers) with regexps.

unary numbers: `7 == '1111111' == '1' * 7`

Primality: recall that positive n is prime if $\neg\exists p, q > 1$ such that $pq = n$.

i.e. `n = ('1' * p) * q, where p, q > 1`

i.e. n matches `(11+)\1+` where `\1` matches expression in ()s.

We can actually write this in terms of a Python regexp:

```
re.match(r'1$|(11+)\1+$', '1'*n)
```

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.
>
> `http://regex.info/blog/2006-09-15/247`

Language Design Issue

The real question we usually want to answer is actually:

"Given a string s and a regex r, find a *substring* in s matching r."

But, which substring should we take? Clearly we want to take the leftmost match. Where do we end the match?

1. *Declarative*: enumerate all matches and return the longest.
2. *Operational*: define how *, | work.

   $e*$ match e as many times as possible while allowing remainder to match.
   $e \mid e$ select leftmost choice while allowing remainder to match.

Example. Consider the regular expression,

```
[a-zA-Z]+ = .* (\\ \n .*)?
```

and the string

```
filesToCompile = a.cpp b.cpp \<\n> d.cpp e.h
```

Exercise. (good practice for exams) Work out the difference between these cases with different semantics.

The semantics of the operational definitions of $*, |$ force the programmer to reason about backtracking, which is non-compositional and carries higher cognitive load.

Note that NFAs can implement the declarative semantics.

Summary.

- We revisited the applications, formulated queries (aka use cases) and looked at example concrete queries.
- We pointed out the different expressive power of FAs versus regexps as implemented in actual languages.
- We explored a language design issue: which semantics of matching to use?

Skills I'd like you to develop:

- figure out which questions to ask when designing a language and the impact of these design choices.
- be able to effectively use an API (we also have this on Lab 1); this API is the regular expression API.