Just Parsin' By...

Due: Wednesday, October 24, 2012 at 11:59pm Pacific
Complete this assignment on your own — 100 total points.

Overview

In this assignment, you'll implement a parser for regular expressions using a technique called "parsing by derivatives". To implement a derivative-based parser for regular expressions, you'll: (a) define the language of regular expressions as an inductive data structure, (b) define the derivative of regular languages as a pattern-matching recursive function, and (c) implement a derivative-based regular-expression parser. The assignment contains some background information on these elements.

Formal Languages, Regular Expressions & Derivatives

An **alphabet** Σ is a set of characters. A **word** w is a sequence of characters $c_1 \cdots c_n$ where $c_i \in \Sigma$; this word has length n. There is a special word called the **null word**, spelled ε; this word has length 0. A **language** L is a set of words.

The set of **regular language**s can be defined inductively as follows:

> The empty language $\varnothing$ is regular.
>
> The language $\{\varepsilon\}$ that contains the null word is regular.
>
> If $c \in \Sigma$, then the language $\{c\}$ is regular.
>
> If L_1 and L_2 are regular, then the language $L_1 \cup L_2$ is regular.
>
> If L_1 and L_2 are regular, then the language $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$ is regular.
>
> If L is regular, then $L^* = \bigcup_{n=0}^{\infty} P^n$ is regular (where $P^0 = \{\varepsilon\}$ and $P^n = P \cdot P^{n-1}$).

The $\cdot$ operator is concatenation. Sometimes the operator is omitted, and juxtaposition implies concatenation, e.g., apple *is the concatenation of five characters.*

The **derivative** of a language L with respect to a character $c \in \Sigma$ is the language of all suffixes in L of words that start with character c. Formally: $\partial_c(L) = \{w \mid c \cdot w \in L\}$. For example, given the language $L = \{\text{apple}\}$ that contains the a single word: $\partial_a(L) = \{\text{pple}\}$, and $\partial_q(L) = \varnothing$. Derivatives are an old idea [1]; they were forgotten by many people for awhile, but have enjoyed a recent resurgence because they can be applied to write simple parsers for regular expressions [2, 3].

You might wonder if you can use derivatives to parse context-free grammars. The answer is yes! See the Might et al. *paper in the References section for more details.*

Regular expressions are closed under derivatives, which means that the derivative of a regular language is also a regular language. The derivatives of regular languages are defined as follows:

> $\partial_c(\varnothing) = \varnothing$
>
> $\partial_c(\{\varepsilon\}) = \varnothing$
>
> $\partial_c(\{d\}) = \{\varepsilon\}$ if $c = d$; otherwise it's $\varnothing$
>
> $\partial_c(L_1 \cup L_2) = \partial_c(L_1) \cup \partial_c(L_2)$
>
> $\partial_c(L_1 \cdot L_2) = \partial_c(L_1) \cdot L_2$ if $\varepsilon \notin L_1$; otherwise its $\partial_c(L_1) \cdot L_2 \cup \partial_c(L_2)$
>
> $\partial_c(L^*) = \partial_c(L) \cdot L^*$

We can use derivatives to parse a word w according to a regular expression L, by chopping the first letter off w and taking the derivative of L with respect to that letter. This derivative will give us a new regular language L_{new}. We then chop off the next letter and take the derivative of L_{new} with respect to that letter. We continue this process until the string is empty. If the final derivative contains the null word, then the original string was in the language described by L; otherwise it wasn't. More formally:

> $\text{parse}(\varepsilon, L) = \varepsilon \in L$
> $\text{parse}(c \cdot w, L) = \text{parse}(w, \partial_c(L))$

1. Implementing a Regular Expression Parser

Implement a regular expression parser using the derivative techniques. In particular, you'll need to:

(a) Define an inductive data structure that captures regular languages. By inductive data structure, I mean that there should be some base objects that correspond to the smallest possible regular languages and some case classes that correspond to the ways that larger regular languages are built from smaller ones.

(b) Define the derivative of a regular language as a function that takes a character and a regular language and results in a regular language.

(c) Define the `parse` function.

2. Feedback *(participation points)*

Answer the questions in the file `feedback.txt`.

References

[1] Brzozowski, J. A. Derivatives of regular expressions. *Journal of the ACM*, 11(4), 481– 494, 1964.

[2] Owens, S., Reppy, J., and Turon, A. Regular-expression derivatives reexamined. *Journal of Functional Programming* 19(2), 173–190, 2009.

[3] Might, M., Darais, D., and Spiewak, D. Parsing with derivatives: a functional pearl. *International Conference on Functional Programming*, 189– 195, 2011.

Materials

The materials are available in the Subversion repository. In your working copy, execute:

```
svn copy https://svn.cs.hmc.edu/dsl/fall12/hw/7 hw7
```

Modify the materials to supply your answers and use the instructions below to submit.

Submitting Your Solution

Make sure you're in the `hw7` directory of your working copy and execute:

```
svn commit
```

Committing is like voting in Chicago: you should do it early and often.

Collaboration and Honor Code

I expect you to abide by the Harvey Mudd Honor code. Your solution to this assignment should be produced by you alone. You may discuss concepts at a high level with any student in the class and / or with the course staff. However, you may not copy solutions from anyone, nor may you search for solutions on the Internet.

If you have any questions about what behavior is acceptable, it is your responsibility to come see me before you engage in this behavior. I will be happy to answer any of your questions.