Programming Languages — Homework 6
Parser combinators

Due: Wednesday, 17 April 2013, 23:55

1 Parser combinators in Java

Starting with the combinator library `ParsersHW.java` on the course web page, you'll implement several new combinators. For each combinator, write test cases to demonstrate that the parsers work correctly.

You should implement all your code for the first part in `ParsersHW.java`, adding a `main` method to test your parser.

Note that `Parsers.java` is also on the web page. It contains more "junk" inserted during class. Do not extend this code to implement your solutions.

1. Write a function `static Parser<String> end()` that returns a parser that succeeds only if the input string is empty. That is, `end().parse("")` should succeed and `end().parse("abc")` should fail without consuming any input.

2. Write a function `static Parser<String> regex(String pattern)` that returns a parser that accepts only strings matching the given regular expression. You can use classes in the `java.util.regex` package.

3. One version of `zeroOrMore` (called `zeroOrMore_Broken` in the code on the web) fails because it calls itself recursively while building the parser:

   ```
   this.then(this.zeroOrMore_Broken())
       .map(ParsersHW.<A>cons())
       .or(empty(Collections.<A> emptyList()));
   ```

   We fixed this in class by introducing a new class, but this solution is rather unsatisfying.

   The problem does not occur in Haskell, because function arguments are evaluated lazily. We can simulate this by passing a *thunk* to `then`, which wraps the recursive call in another object.

   Define an abstract class `Thunk<A>` with two methods: the abstract method `compute` performs the actual computation, returning an `A`; the non-abstract method `force` calls `compute` but caches the result so that subsequent calls to `force` do not call `compute` again.

   Define a method `lazyThen` in `Parser` than is like `then`, but takes a `Thunk<Parser<B>>` instead. The thunk should not be forced until needed. You will not be able to reuse the existing `SequenceParser` to implement `lazyThen`.

   Define the method `zeroOrMore` which fixes the problem in `zeroOrMore_Broken` using `lazyThen`. Try to have as few differences as possible with the implementation of `zeroOrMore_Broken`.

4. Add the method `oneOrMore` to the `Parser` class. This should return a parser that accepts one or more occurrences of the string parsed by the receiver (i.e., `this`).

   Full credit will be given only if you implement the new parsers by combining existing parser combinators. Your implementation should not explicitly create a subclass of `Parser` (including an anonymous subclass). Indeed, you should not have to deal directly with `Result` values at all.

5. Add the following two methods to the `Parser` class:

   ```
   Parser<List<A>> zeroOrMore(Parser<?> sep)
   Parser<List<A>> oneOrMore(Parser<?> sep)
   ```

These methods *overload* the existing methods of the same name but take an additional argument. The `zeroOrMore` (resp., `oneOrMore`) methods should return a parser that accepts zero (one) or more occurrences of the string parsed by the receiver (i.e., `this`), each of which is separated by the string parsed by `sep`. For example, if `integer` is a `Parser<Integer>` that accepts integer literals, and `comma` is a parser that accepts the comma character, then `integer.zeroOrMore(comma)` should accept a comma-separated list of integers. The type of this parser should be `Parser<List<Integer>>`. The separator is parsed, but its result is discarded.

Full credit will be given only if you implement the new parsers by combining existing parser combinators. Your implementation should not explicitly create a subclass of `Parser` (including an anonymous subclass). Indeed, you should not have to deal directly with `Result` values at all.

6. The parser implementation we've built is rather inefficient. When a branch of an `or` fails, the parser backtracks to try the other branch. This can duplicate work: for example, `p.then(q).or(p.then(r))` will invoke the `p` parser twice if `p` is successful and `q` fails.

   To address this, we can add *memoization* to the parser. That is, results are cached so that if the parser is invoked more than once on the same input, the previous result is returned immediately rather than rerunning the parser.

   Add a method `Parser<A> memo()` to the `Parser` class which returns a parser that behaves the same as `this` except it memoizes its results.

2 Parsing dates

The Unix `date` command outputs the date and time as follows:

```
Thu Apr 11 00:09:59 CEST 2013
```

Write a `Parser<java.util.Date>` that parses one of these date strings and returns a `java.util.Date`.

Implement a `main` method that reads a text from standard input, and for each line uses the parser to build a `Date` object `d` from that line, and then prints `d.toString()`. If the line text is not contain a legal date, print "`fail`" for that line.

You should use the parser combinators defined `ParsersHW.java`, but implement a new `main` method in a separate class `ParseDate` in its own file.

Submission

1. Complete the survey linked from the course web page after completing this assignment.
2. Submit your code and solutions on Moodle by 23:55 on Wednesday, 17 April 2013. Include your name in each file you submit.