Lecture 13, 16th May 2006

- Finite State Machines revisited
  - Adding a class for DFSMs
    - as sibling of NFSM, with new abstract superclass
      In class, we used the refactoring browser to help create the sibling class. Looked at each method in NFSM, and decided if it should be promoted to the superclass, or whether a new (and different) version should be created in the DFSM subclass.

- as subclass of NFSM. DFSMs are, after all, a specialization of NFSMs. I don't think that there is anything wrong with this, even though we chose to pursue the above alternative in class. However, if you pursue this one, it's really important to cancel (= override with self shouldNotImplement) any methods that are not appropriate.

- Simple and clean way of writing the DFSM>>newFrom: aNFSM method
  - Can't really disregard efficiency when we changing the "big O" runtime.

- Smalltalk Parsers
  - Backing-up after a failed parse
    - the primitive parsers maintain the invariant:
      - if they fail, they do not advance the underlying stream
      - if they succeed, the stream is advanced by an amount that corresponds to what was parsed.
    - the parser combinators
  - However, the combinators don't:
    - the +++ (biased OR) combinator is not a problem, because the first alternative to succeed wins, and the previous ones, having failed, will not have advanced the input.
    - >>= is a problem though. in a >>= b, if the parser a succeeds, and the parser b fails, the whole sequence is deemed to fail, and the input accepted by a must be "backed out"
      - There are two problems:
        - the >>= combinator does not know how much input its left argument has swallowed
        - we were representing parsers as blocks (instances of BlockContext), so there is no (clean) way to access the stream in order to back it up.
      - The obvious solution would be to subclass BlockContext, to enable additional operations to access the stream, both to record the position and to restore it.
    - However, this won't work, because blocks are created by a special compiler syntax, and there is no way to create an instance of a subclass. (The same is true of SmallInteger).
  - Instead we use Delegation.
    - We create a new class of objects, called ParserFuns, that
      - have an instance variable that is a block, which, when executed, will parse, and
      - have an instance variable that references the stream over which the parser operates.
    - We add one operation to BlockContext, asParserOn: aStream builds the ParserFun object and initializes the two instance variables.
    - We give ParserFuns a method parse, that sends value to the stored block. (We decided that parse was a more descriptive and less confusing name than value)
    - We started doing this refactoring in class, and got almost all of the way through. We simply ran the tests, got a Message Not Understood because we were still sending value where we needed to send parse, or because a ParserFun didn't yet support the right operations, and implemented what we needed.
  - When we had all of the ParserFun machinery in place, we were able to easily implement the backup after a failed parse. Here is the code:
When we had all of the ParserFun machinery in place, we were able to easily implement the backup after a failed parse. Here is the code:

```
ParserFun >>>= aOneArgumentBlock
   "sequencing.
   Answers a parser that, when run, will first run me, and then evaluate
   aOneArgumentBlock. with the result of running me as its argument,
   and then finally run the answer obtained from evaluating the block,
   which is assumed to be a parser."
^ [! start |
   start := parserStream position.
   self parse
   ifNotNilDo: [:v | (aOneArgumentBlock value: v) parse
   ifNil: [parserStream position: start. nil]]]
   asParserOn: parserStream
```

The saving of the start position and the restoring of it after a failure is all that's needed.

After class, I added a third instance variable to ParserFun: a name. This is nothing more than a string that helps in debugging.

For example:

```
ParserStream>>identifier
   ^ self lower
   >>= [x | self alphaNum many
   >>= [xs | self optionalSpaces
   >> (self unit: (xs copyWithFirst: x))]
   name: 'identifier'
```

When I look at this parser in the debugger, I can now see that it's a parser for an identifier, rather than a meaningless jumble of nested blocks.

This example also shows the >>= combinator, which takes left and right arguments that are both parsers, and runs them in sequence. So, it is like >>>=, but without the provision for piping the result of the first parse into the second.

In the example, we are not concerned with the result of optionalSpaces, so there is no need to bind it to a variable.

The code for this version is on the website.