Compositional Parsers in Smalltalk using ParserFun objects
Recap

- Where were we on Monday?
  - CS510ap-Parsers-apb.1.2.mcz on SqueakSource
    - Parsers were blocks…
    - created by methods on a ParserStream
      - the stream was captured implicitly in the environment of the block
      - combinators (|, >>, star, plus) were operations on blocks
      - failure of a parse was represented as nil
The Good

• We had some parsers that worked
• two ways of capturing parse results
  ◦ concatenation (plus, star)
  ◦ >>= , which binds the result of the left parser to the argument of the block that is its right-argument

  identifier
  "answers the parsed identifier"
  ↑ self lower
  >>= [:x | self alphaNumeric star
  >>= [:xs | (self return: x,xs )]]
The Bad

- couldn’t maintain the invariant that a failing parser does not consume the input
  - lhs of >>= is a block

- couldn’t write operations like option, applicable to any parser, in a compositional way
  
  ```
  option
  "zero or one applications of this parser. Always succeeds."
  ↑ self | <what?> epsilon
  ```

- in both cases, we need explicit access to the input stream
What have we learned?

• Blocks are good
  ▶ let us compose parsers with \|, execute them with \texttt{value}

• Blocks are not enough
  ▶ we also need access to the stream

• Debugging is hard
  ▶ What was that parser what just failed?
Now that we know more…

• we are ready for a major refactoring
  ▶ ParserFun is a new class of parsers
    ◦ instance variables parserBlock and name
  ▶ parse: takes the input stream as argument
    parse: aStream
    "run me as a parser, by executing my parserBlock
    with aParserStream as argument."
    ↑ parserBlock value: aStream
many class-side methods to create new parsers

\[
\text{ParserFun letter} \\
\text{ParserFun digit} \\
\text{ParserFun satisfies: aPredicate}
\]

- parsers no longer capture the input stream, so they are constants

\[\text{star, |, >>=, token}\] are instance-side methods that operate on ParserFuns and answer new ParserFuns
• ParserFuns created by

ParserFun
  named: 'aMnemonicName'
  doing: [ : stream | ... parse actions on stream ]

• … or by a shortcut operation on a block

fail
"The parser that, when evaluated, does nothing
and always fails"
↑ [nil] asParserNamed: 'fail'

which is implemented by sending
ParserFun named:doing:

• ParserFun new is cancelled
• we can correctly back-up after a failed parse

```smalltalk
>>= aOneArgumentBlock
"sequencing…"
↑ [ :pStr | | start |
start := pStr position.
(self parse: pStr) ifNotNilDo: [:v | ((aOneArgumentBlock value: v) parse: pStr)
ifNil: [pStr position: start. nil]]]
```

asParserNamed: self name , '>>=' , '... '
• we can write combinators like token

  token

  "a Parser that applies this parser, and, if I succeed, consumes any junk that follows. Answers whatever I answer"

  ↑ self >>= [ :result | ParserFun junk >>

  (ParserFun return: result) ]

  name: self name, '-token'

• the names help us to figure-out what parser was running when we find a bug
Issues

• getting the results of the parse:
  ▶  >>= operator lets us bind the result of the lhs …

    keyword := (ParserFun string: 'if') | (ParserFun string: 'then') | (ParserFun string: 'else') >>= [:r | ParserFun spaces >>= (ParserFun return: r)] name: 'keyword'.

    … but it’s pretty messy

• >>= operator is like >>= but discards the result of the lhs, takes parser, not block on rhs
• hard to keep track of what the results are going to be

  - I adopted the “sequence convention”:
    - results are *always* a sequence, and combinators concatenate sequences.
    - so, ParserFun char: $a$ now answers a (unit) sequence of characters, 'a', not a single character.
    - ParserFun letter answers a (single character) string, and ParserFun letter plus a (possibly) longer string.
    - ParserFun identifier answers a (unit) sequence of symbols, and ParserFun identifier plus a (possibly) longer sequence of symbols
The `asString` combinator

- This meant changing the result of many primitive parsers from character to unit string
- Capture the pattern as a combinator:

```
asString
  "run myself, assuming that I return a character. Convert it to a string."
  ↑ self >>= [ :c | ParserFun return: c asString ]
```
Comma vs. >>=

• Compare:

\[ \text{ldorll} := \text{ParserFun letter} >>= [\text{:c} | \text{ParserFun digit} >>= [\text{:d} | \text{ParserFun return: c,d}]] | (\text{ParserFun letter} >>= [\text{:c} | (\text{ParserFun letter} >>= [\text{:d} | \text{ParserFun return: c,d}]]) \]

and

\[ \text{ldorll} := (\text{ParserFun letter, ParserFun digit}) | (\text{ParserFun letter, ParserFun letter}) \]

• Of course, if concatenation is not what you want, this won’t help
What about these?

- **BNF:**
  
  \[
  \text{number ::= digit number}^* 
  \]

- **>>= style:**
  
  \[
  \text{number ::= ParserFun digit} \gg= [\text{:d} | \\
  \text{number star} \gg= [\text{:num} | \\
  \text{ParserFun return: d, num}]]. 
  \]

- **comma style:**
  
  \[
  \text{number ::= ParserFun digit , number star.} 
  \]
The Code

- CS510ap-Parsers-apb.5 in SqueakSource.
- need to load NewCompiler (copy in SqueakSource) and turn on preferences compileUseNewCompiler and compileBlocksAsClosures
- If you have trouble, try loading ImageFixes-apb.?