Parser Combinators

in Smalltalk
Tim’s Trick

- What’s so cool about functional programming?
  - referential transparency
  - if $x = foo \text{ stuff}$, then
    $$x + x = foo \text{ stuff} + foo \text{ stuff}$$
  - $=$ is mathematical equality, and you can replace equal by equal
What about effects?

• Suppose that $x$ is not just a value but has an effect:

\[
\text{if } x = \text{nextInt input}, \text{ then }
\]

\[
x + x \neq \text{nextInt input} + \text{nextInt input}
\]

• In the presence of effects, you can’t replace equals by equals

• Say bye bye to equational reasoning
How does Haskell combine equational reasoning and effects?

- Monads separate the *definition* of commands from the *execution* of commands
  
- Command definition is still referentially transparent

- Command execution is “special”
  
  - *do* syntax
What about other languages?

• Can we separate command definition and command execution in Java, or in Smalltalk?

• Yes!
  
  • but the defaults are backwards compared to Haskell
  
  • methods can be executed (= do),
    
    • but that’s the default!
    
    • How can they be manipulated?
We have sequential composition

• BNF:

\[
\text{ifStmt} ::= \text{if boolean then } \text{stmts* else } \text{stmts* fi}
\]

• code:

```
parseIfStmt
    parseKeyword: #if.
parseBoolean.
parseKeyword: #then.
parseStatements.
parseKeyword: #else.
parseStatements.
parse: #fi
```
but that’s not enough

• BNF:

  \[ \text{stmts}^* ::= \epsilon \mid \text{stmt} \ \text{stmt}^* \]

• code:

  parseStatements

    ??
How can we define Squeak combinators for Parsers?

- make the parsers *objects*, not methods

- the parser objects can be
  1. *run*, to perform the parse, or
  2. *combined*, to make more complex parsers
First attempt

• What kind of Squeak object will perform an action when run?

• blocks
What about the input?

- We need a way of representing a sequence of objects along with the current position in that sequence
- Streams provide exactly that functionality
- Compare Streams with Iterators
Iterating with do: and with a Stream

• initialize the collection

\[
c := 'abcde'.
\]

• with an internal iterator:

\[
c \text{ do: } [ : \text{each} \mid \text{Transcript show: each } ]. \text{Transcript cr.}
\]

• with a Stream (external iterator)

\[
s := \text{ReadStream on: c.} \\
[ s \text{ atEnd } ] \text{ whileFalse: } [ \text{Transcript show: s next } ]. \text{Transcript cr.}
\]
Implementing Parsers

• A parser is implemented as a block.
• The *result* of evaluating the block is:
  - if the parse succeeds: a sequence of correctly parsed items
  - if the parse fails: nil.
• The *effect* of evaluating the block is to:
  - advance the underlying stream if the parse succeeds, and
  - not to advance it if the parse fails.
• This invariant must be maintained across compound parsers.