Parser Combinators

in Smalltalk



Tim's Trick

- What's so cool about functional programming?
 - referential transparency
 - if x = foo stuff, then x + x = foo stuff + foo 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:

```
if x = nextInt input, then
x + x \quad nextInt input + 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 Monads
- 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:

ifStmnt ::= if boolean then stmts* else stmts* fi

code:

```
parselfStmnt
parseKeyword: #if.
parseBoolean.
parseKeyword: #then.
parseStatements.
parseKeyword: #else.
parseStatements.
parseStatements.
parse: #fi
```



but that's not enough

• BNF:

```
stmts^* ::= \epsilon \mid stmt \ 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

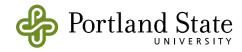
```
c := 'abcde'.
```

with an internal iterator:

```
c do: [: each | Transcript show: each ]. Transcript cr.
```

with a Stream (external iterator)

```
s := ReadStream on: c.
[ s atEnd ] whileFalse: [ Transcript show: s next ].
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.

