CSCI 334:
Principles of Programming Languages

Lecture 16: Parsing

Instructor: Dan Barowy
Williams

1

Topics

Parser combinators

2

Your to-dos

1. Read Parser Combinators if you have not already done so.
2. Lab 8, **due Monday, April 22** (partner lab).
3. Project **checkpoint #1**, due **Monday, April 29**.

3

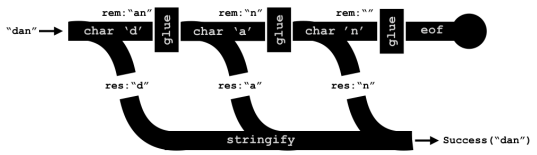
Announcements

- **Midterm exam**, in class, Thursday, May 2.
- Colloquium: **Pre-registration info session** Friday, April 19 @ 2:35pm in Wege Auditorium.
  - Learn about Computer Science courses offered Fall 2024.
  - Talk to professors about their classes.
  - Discuss CS major declaration.
  - Meet other Computer Science students.

4

Parser Combinators

5



6

Recall: mental model is of pipes. A parser is a pipe. Parser combinators MAKE sections of pipe or glue them together.

Basic Primitives

- Input

```
type Input = string * int * bool
```

- Output

```
type Outcome<'a> =
| Success of result: 'a * remaining: Input
| Failure of fail_pos: int * rule: String
```

7

Definitions of Input and Output.

Basic Primitives

- A parser is

```
type Parser<'a> = Input -> Outcome<'a>
```

- Keep in mind: a parser *is a function*.

8

A Parser is a function from Input to Output. The ‘a represents the type of data returned, which is configurable.

Two varieties of parser

- Parsers that consume input. Correspond with grammar terminals.
- Parsers that combine parsers. Correspond with grammar non-terminals. Also called “combining forms.”
- For flexibility, you can also have parsers that do both.

9

Terminal parsers

```
pchar(c: char): Parser<char>
```

```
> let input = prepare "ddd";;
val input: Input = ("ddd", 0, false)

> let d = pchar 'd';;
val d: Parser<char>

> d input;;
val it: Outcome<char> = Success ('d', ("ddd", 1, false))
```

10

Here's an example of using a basic parser, d, that consumes input. Observe that d is the parser. pchar 'd' is the function that MAKES the parser.

Combining parsers

```
pseq
   (p1: Parser<'a>)
   (p2: Parser<'b>)
   (f:'a -> 'b -> 'c)
   : Parser<char>
```

```
> let dd = pseq d d (fun (x,y) -> (string x) + (string y));;
val dd: Parser<string>

> dd input;;
val it: Outcome<string> = Success ("dd", ("ddd", 2, false))
```

11

Here's an example of a parser, dd, made by running the d parser twice in sequence. The pseq function glues them together. dd runs the first d, and if that is successful, runs the second d. If the second is successful, it takes the two outputs as a tuple and gives them to the function f. In our example, we convert x and y to strings, then concatenate them. (try removing the 'string' functions and see what you get)

Combining parsers

- pseq :
  p1:Parser<'a>
  ->
  p2:Parser<'b>
  ->
  f:('a * 'b -> 'c) -> Parser<'c>
- p1 is a parser.

12

Combining parsers

13

- `pseq :`
  `p1:Parser<'a>`
  `->`
  `p2:Parser<'b>`
  `->`
  `f:('a * 'b -> 'c) -> Parser<'c>`
- **`p2`** is a parser.

Combining parsers

14

- `pseq :`
  `p1:Parser<'a>`
  `->`
  `p2:Parser<'b>`
  `->`
  `f``:('a * 'b -> 'c) -> Parser<'c>`
- **`f`** is a function that takes the result of **`p1`** and **`p2`** and **does something** with it. That **something** is up to **you**.

Let's try it

15

- `pseq (pchar 'z') (pchar 'o') id`
- `id` is F#'s identity function.
- Let's play with this in `fsharpi`.

Try this one on your own. You can load the combinator library in dotnet fsi like so.

1. Put Combinator.fs in your current directory:

```
$ wget https://williams-cs.github.io/cs334-s24-www/assets/code/Combinator.fs.txt
$ mv Combinator.fs.txt Combinator.fs
```

2. Open the F# REPL:

```
$ dotnet fsi
```

3. Load the Combinator library. This will print lots of stuff if successful.

```
> #load "Combinator.fs";;
```

4. Open the Combinator library.

```
> open Combinator;;
```

5. Now go ahead and use the library, e.g.,

```
> let myparser = pseq (pchar 'z') (pchar 'o') id;;
val myparser: Parser<char * char>
> myparser (prepare "zoo");;
val it: Outcome<char * char> = Success (('z', 'o'), ("zoo", 2,
```

More details

16

- It is **critical** that you read the “Parser Combinators” reading.
- I suggest that you **sit down, uninterrupted, for an hour or two**, and **work through the examples** in `fsharpi`.
- The reading builds the `Combinator.fs` library that you are given for HW8.

Example: brace language

17

Here’s a language definition in plain English.

- An *expression* is a sequence of *terms*, consisting of *at least one term*.
- A *term* is either '`aaa`', '`bbb`', or a *brace expression*.
- A *brace expression* is '`{`', followed by an *expression*, followed by '`}`'.

Example: brace language

```
<expr>  ::= <term>+
<term>  ::= aaa
          |  bbb
          |  <brace>
<brace> ::= { <expr> }
```

Let's write a parser for this language.

18

Here's the same definition in Backus-Naur Form. Let's implement this. See the bracelang project for sample code.

If you're looking for a nice practice problem, here's a good one. Write a parser for this language. The start symbol is <expr>.

<expr> ::= <two>*

<two> ::= aa | bb | cc

Recap & Next Class

Today:

Parser combinators

Next class:

Program evaluation

19