CSCI 334:
Principles of Programming Languages

Lecture 15: Parsing, part 2

Instructor: Dan Barowy
Williams

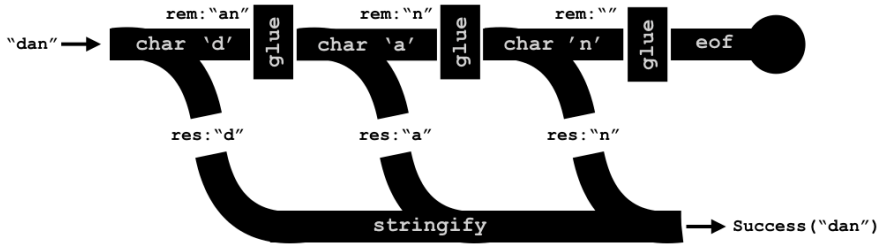
Topics

Using parser combinators

Your to-dos

1. Quiz, **due Wednesday 11/8**.
2. Lab 9, **due Sunday 11/13** (partner lab)

Parser Combinators



Parser Combinators

- A kind of recursive decent parser.
- A **recursive descent parser** is a parser built from a set of **mutually recursive procedures** where each such procedure usually **implements one of the productions** of the grammar.
- Recursive descent parsers are "**top-down**," meaning that they recognize sentences by expanding nonterminals, starting from the start symbol.
- "**Bottom-up**" parsers start with *terminal* symbols and work in the opposite direction, often utilizing dynamic programming… these are more common in practice!

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
```

Basic Primitives

- A parser is

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

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

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.

A very simple terminal parser

- To parse a given char

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

- Notice that the generic type inside `<brackets>` is the return type of the parser.
- So `pchar` returns a *parser*.
- When it is run with an *input*, it returns an `Outcome<char>`.

How to use it

- `(pchar 'z') input`
- `input` must be "prepared" first.
-
```
> let input = "zoo";;
val input : string = "zoo"
```
-
```
> let i = prepare input;;
val i : Input = ("zoo", true)
```
-
```
> (pchar 'z') i;;
val it : Outcome<char> = Success ('z',("oo", true))
```

A very simple combining parser

- To parse two things in sequence:

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

- It looks more complicated than it is.
- Let's look at each part.

A very simple combining parser

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

A very simple combining parser

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

A very simple combining parser

- `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**.

How to use it

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

More details

- 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 `Parsers.fs` library that you are given for HW9.

Example: brace language

- 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> }
```

We will write a parser for this language.

Recap & Next Class

Today:

Writing a parser

Next class:

Building an entire language