CSCI 334:
Principles of Programming Languages

Lecture 12: Parsing

Instructor: Dan Barowy

Williams

Topics

Parser combinators

Your to-dos

1. **Trip to WCMA on Thursday**. Go directly there. Be sure to bring your notes from last time.
2. Read *Gumptionology 101* and *Beating the Averages* for **this week's quiz** (take home).
3. Lab 7, **due Wednesday, April 9** (partner lab). This is **project checkpoint #1**!
4. Read *Parsing* for next week's lab.

Announcements

- Class of 1960's speaker series: **Tim Kraska, MIT**!
- *Generative AI is Overhyped But Real*, **Thurs, Apr 10 @ 7:30pm in Wachenheim Auditorium**
- *ML and Generative AI for Data Systems*, **Thurs, Apr 11 @ 2:35pm in Wege Auditorium**

Tim Kraska is an Associate Professor of Electrical Engineering and Computer Science in MIT's Computer Science and Artificial Intelligence Laboratory, co-director of the new GenAI Impact Initiative at MIT, a director of applied science at Amazon Web Services (AWS), and was a co-founder of Instancio and of Einblick Analytics (both acquired). Currently, his research focuses on using ML/GAI for (data) systems and agentic systems. Before joining MIT, Tim was an Assistant Professor at Brown and spent time at Google Brain. Tim is a 2017 Alfred P. Sloan Research Fellow in computer science and received several awards including the VLDB Early Career Research Contribution Award, the Intel Outstanding Researcher Award, the VMware Systems Research Award, the university-wide Early Career Research Achievement Award at Brown University, an NSF CAREER Award, as well as several best paper and demo awards.

Announcements

Please **consider being a TA** next semester
(especially for this class!)

Applications **due Friday, April 18**.

https://csci.williams.edu/tatutor-application/

Resubmissions

Resubmission procedure

The goal of this
course is **skill mastery**.

Resubmission procedure

Allows you to earn **up to 50%
of the lost points**.

E.g., **if you got a 50%** on the midterm,
you can get a 75% on resubmission.

Each midterm is 20% of your final grade.
This is worth doing!

Resubmission procedure

1. You have **until the end of reading period**.
2. Resubmission **must include both** the **original work** and the **new submission**.
3. Must be accompanied by an **explanation document**, written in plain English.
4. You **cannot resubmit** something you never did.

Resubmission procedure

Explanation document **must identify**:

1. **What** the mistake is.
2. **How** you fixed the mistake.
3. **Why** the new version is correct.

Resubmission procedure

Resubmit code **electronically**
(i.e., using git).

Resubmit exam **on paper**
(i.e., hand it to me or put in dropbox).

Handwritten resubmissions
will not be accepted.
Type them and print them out.

Resubmission procedure

Sample from CS334:

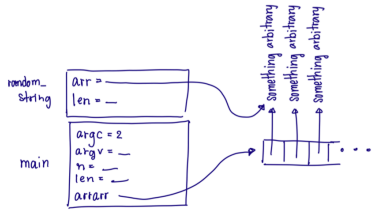
2. Troubleshooting
My fix was slightly wrong. Righr before calling $random\_string()$, I added

```
char * arrarr[i] = malloc(sizeof(char)*MAXLEN);
```

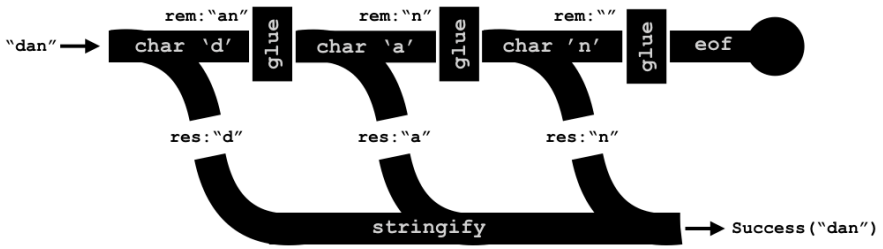
when what I should have added is

```
arrarr[i] = malloc(sizeof(char)*MAXLEN);
mcheck(arrarr[i]);
```

There is no need for "char *" because I am not declaring $arrarr$.
I got my explanation and drawing wrong. In my drawing, I had $arrarr[i]$ pointing back to a call stack because I thought the program would automatically allocate memory on a call stack if we did not $malloc()$. What I should have said is that without allocating sub-array $arrarr[i]$, the address currently living in the sub-array is arbitrary so the value referred to by the sub array is also arbitrary. When we call $memset()$ or manipulating $arrarr[i]$ in $random\_string()$, we are likely to get memory errors. Below is what I should have drawn.



Parser Combinators



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.
- For flexibility, you can also have parsers that do both.
- *Parser combinators are themselves a mini programming language.*

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))
```

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))
```

Combining parsers

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

Combining parsers

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

Combining parsers

- `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

- `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 walks you though coding up the `Combinator.fs` library that you are given for Lab 8.

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> }
```

Let's write a parser for this language.

Recap & Next Class

Today:

Parser combinators

Next class:

WCMA field trip #2