CSCI 334:
Principles of Programming Languages

Lecture 13: Parsing

Instructor: Dan Barowy

Williams

Announcements

HW7 Q5 updated (it's a tad easier now)

You can find the Parser lib linked in the reading.

HW4 resubmission: last day to submit is today

HW8 is your project proposal: think about who you want to work with. I am happy to help find partners.

Compound Types:

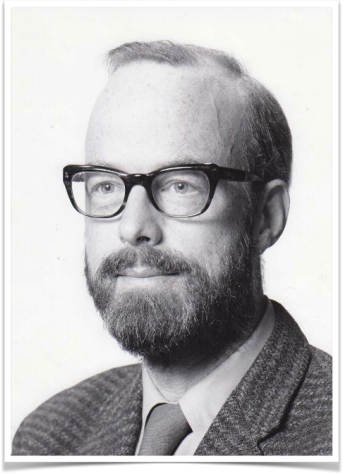
Records, ✓
Lists, ✓
Tuples, ✓
ADTs

Algebraic Data Types

not
Abstract Data Types

Hoare Property

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies." — C.A.R. Hoare



Hoare Property

ADTs make the structure of a program's logic "more obvious."

ADT (Java)

```
public static final int NORTH = 1;
public static final int SOUTH = 2;
public static final int EAST = 3;
public static final int WEST = 4;

public move(int x, int y, int dir) {
  switch (dir) {
    case NORTH: ...
    case ...
  }
}
```

ADT (F#): "discriminated unions"

```
type Direction =
     North | South | East | West;

let move x y dir =
  match x,y,dir with
   | x,y,North -> x,y-1
   | x,y,South -> x,y+1
```

- Above is an "incomplete pattern"
- ML will warn you when you've missed a case!
  - "proof by exhaustion"

ADTs can have parameters

```
type Shape =
| Rectangle of float * float
| Circle of float
```

- Pattern match to extract parameters

```
let s = Rectangle(1.0,4.0)
match s with
| Rectangle(w,h) -> …
| Circle(r) -> …
```

ADTs can have named parameters

```
type Shape =
| Rectangle of width: float * height: float
| Circle of radius: float
```

- Names are useful for initialization and documentation.

```
let s = Rectangle(height = 1.0, width = 4.0)
```

ADTs can even be recursive and generic

```
type MyList<'a> =
| Empty
| NonEmpty of head: 'a * tail: MyList<'a>
```

```
> NonEmpty(2, Empty);;
val it : MyList<int> = NonEmpty (2,Empty)
```

handling errors with ADTs and patterns

- Another example: handling errors.
- F# has exceptions (like Java)
- But an alternative, easy way to handle many errors is to use the `option` type:

```
type Option<'a> = None | Some of 'a
```

handling undefinedness with patterns

Write a function `get_nth` that takes a list of strings and an `int n` and returns the n^{th} element of the list, where the head is 1^{st}.

handling undefinedness with patterns

```
let rec get_nth xs n =
  match xs,n with
  | []     , _ -> None
  | (x::xs), 1 -> Some x
  | (x::xs), n ->
    if n > 1 then get_nth xs (n-1) else None
```

option type

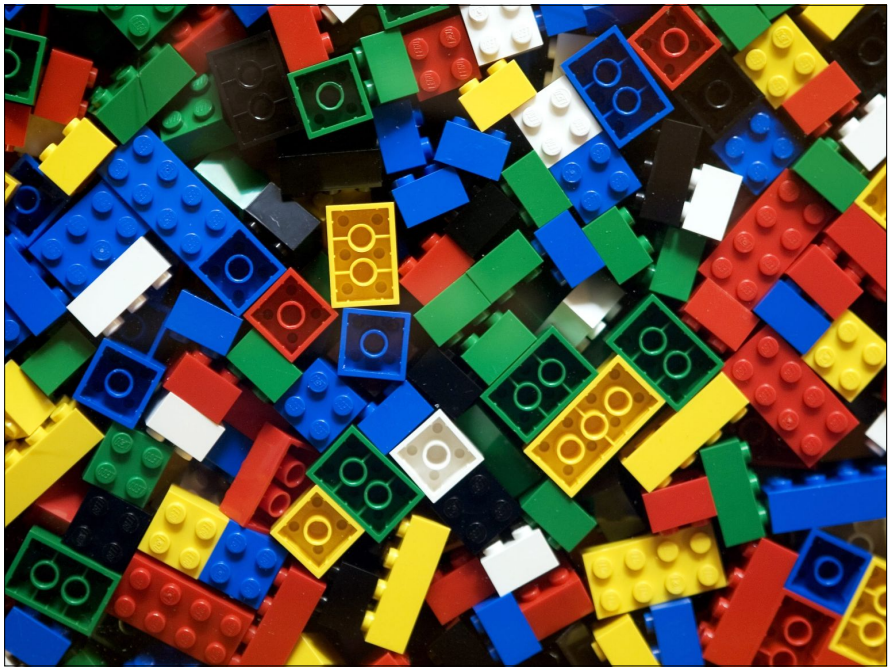
- Why `option`?
- `option` is a data type;

  not handling errors is a static type error!

handling errors with patterns

```
> get_nth [1;2;3;4] 3;;
val it : int option = Some 3
> get_nth [1;2;3;4] 0;;
val it : int option = None
> get_nth [1;2;3;4] 5;;
val it : int option = None
> get_nth [1;2;3;4] -2;;
val it : int option = None
```

Parser Combinators



Parser Combinators

- A kind of recursive decent parser.
- Recursive descent parser:
  A top-down parser built from a set of mutually recursive procedures where each such procedure usually implements one of the productions of the grammar.

Basic Primitives

- Input

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

- Output

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

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.

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 `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 HW7.

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 '`}`'.