Parsers

What is a parser?

- A function to extract or decode an interesting value from a string

Start with identity function - returns its argument:

```
\x -> x
```

This could serve as a parser that returns the entire string. Not especially useful, but a good start.

- It exists in Elm as `identity` so we could define a parser as:

  ```
  parser : String -> String
  parser = identity
  ```

More general parsing

In general, however, we may want to:

1. match only specific patterns -- parsing might fail
2. convert the matching string to some other type, e.g. abstract syntax tree
3. not match the entire string -- some will be left over

E.g.: say we're interested in the fist character of the input string:

- returns `Char`, not `String`
- also return the unparsed input in a tuple

  ```
  char0 (c::cs) -> (c, cs)
        []      -> -- FAILURE!
  ```

Handling failure

- This will fail if the input string is empty:
  - Hutton and Meijer use List
  - elm-tools/Parser uses Result
  - I'm using `Maybe`; keeps the examples short (no error messages) -- less to type.

Parse the first character

Note: Strings aren't *natively* `List Char` in Elm:

```
character0 : String -> Maybe (Char, String)
character0 inp =
    case String.toList inp of
        [] ->
            Nothing

        head :: tail ->
            Just ( head, String.fromList tail )
```

- Turns out we can use [uncons]:

  ```
  character0 : Parser Char
  character0 = String.uncons
  ```

Generalising `parser type`

We don't always want `Char`: use a type variable:

```
type alias Parser a
    = String -> Maybe (a, String)
```

Now:

```
character0 : Parser Char
character0 = String.uncons
```

- try it

  ```
  > character0 "abc"
    Just ('a',"bc") : Maybe ( Char, String )
  ```

Running parsers

`run` ignores any unconsumed input and returns the current value:

```
run : Parser a -> String -> Maybe a
run p s =
    p s |> Maybe.map Tuple.first
```

- Now we can *run* `character0` on a given string

  ```
  > run character0 "abc"
    Just 'a' : Maybe Char
  ```

Pattern matching

What if we want to:

- match only a particular character or class (digits or letters, etc.)?
- convert to other types.
- fail if not all the input sting was matched?

E.g. match only digits (`0-9`) :

Parse a character with `character0` **and then**: **succeed** if that character was a digit, **fail** otherwise?

Combining parsers

```
andThen : (a -> Parser b) -> Parser a -> Parser b
andThen continuation parser =
    \inp ->
        case parser inp of
            Just ( aVal, rest ) ->
                continuation aVal rest

            Nothing ->
                Nothing
```

- Because `Parser = String ->; Maybe (a, String)` and `continuation: a ->; String ->; Maybe (b, String)`

  If we `uncurry` continuation, its type becomes: `(a, String) ->; Maybe (b, String)` and we can reuse `Maybe.andThen`:

```
    andThen_ : (a -> Parser b) -> Parser a -> Parser b
    andThen_ continuation parser =
       parser >> Maybe.andThen (uncurry continuation)
```

- `andThen` is *bind* (`>;>;=`), with its arguments flipped: Elm style encourages use of `|>;` pipes

```
      (|>) : a -> (a -> b) -> b
```

- So now we can write:

```
     character0
     |> andThen (\c -> -- do something with char c... )
```

- What we want to do is test if `c` is a digit:

```
      character0
         |> andThen
             (\c ->
                 if Char.isDigit c then
                     -- parsing succeeds, and the result is c
                 else
                     -- parsing fails
             )
```

Always succeed

A parser that never fails, and always returns a specific value `val`, and the entire unconsumed input `inp`

```
succeed : a -> Parser a
succeed val =
    \inp -> Just ( val, inp )
```

Always fail

Ignore the input and just fail

```
fail : Parser a
fail =
    \inp -> Nothing
```

Digit parser

```
digit : Parser Char
digit =
    character0
        |> andThen
            (\c ->
                if Char.isDigit c then
                    succeed c
                else
                    fail
            )
```

Satisfies

Generalising: `Char.isDigit` becomes any predicate on `Char`:

```
satisfies : (Char -> Bool) -> Parser Char
satisfies predicate =
    character0
        |> andThen
            (\c ->
                if predicate c then
                    succeed c
                else
                    fail
            )
```

- Rewrite `digit`:

```
digit : Parser Char
digit =
    satisfies Char.isDigit
```

Other Char parsers

Now we can match specific characters or character groups:

```
upper : Parser Char
upper =
    satisfies Char.isUpper

char : Char -> Parser Char
char c =
    satisfies (\x -> x == c)
```

Convert to other types: intDigit

What if we want results of some other type than Char? E.g.: parse a digit and then convert it to `Int`.

```
toInt : Char -> Int
toInt c =
    Char.toCode c - Char.toCode '0'


intDigit : Parser Int
intDigit =
    digit
        |> andThen
            (\a ->
                succeed (toInt a)
            )
```

- Or, point free:

```
intDigit : Parser Int
intDigit =
    digit |> andThen (succeed << toInt)
```

Map

Generalising:

- `digit` becomes any parser `p :  Parser a`
- `toInt` becomes any function `f : a ->; b`

```
map : (a -> b) -> Parser a -> Parser b
map f p =
    p |> andThen (\a -> succeed (f a))
```

- Or, point free:

  ```
  map : (a -> b) -> Parser a -> Parser b
  map f =
      andThen (succeed << f)
  ```

- And we an rewrite `intDigit` using map:

  ```
  intDigit : Parser Int
  intDigit =
      digit |> map toInt
  ```

What if we want the multi-digit version of this?

Take

`take` maps a function wrapped in a parser (parsers are applicative functors):

```
take : Parser a -> Parser (a -> b) -> Parser b
take pa pf =
    pf |> andThen (\f -> map f pa)
```

Take has its its arguments flipped for *infix* use in `|>;` pipelines.

To use it we need a *function parser* `pf : Parser (a ->; b)`: a parser that returns a function:

- Use `succeed`. E.g. make a parser that returns `toInt`:

  ```
  > succeed toInt
    <function> : Parser.Parser (Char -> Int)
  ```

- Now we can rewrite `intDigit:

  ```
  intDigit1 : Parser Int
  intDigit1 =
      succeed toInt
          |> take digit
  ```

- Or use a pipe :

  ```
  > run (succeed List.singleton |> take digit ) "50"
  Just ['5'] : Maybe (List Char)
  ```

Partial application

Take is more interesting with multi-argument, curried functions.

E.g.: `(+) : number ->; number ->; number`

- Partially applying this, with `take`, is another way to get a function parser:

  ```
  > succeed (+)
    <function> : Parser.Parser (number -> number -> number)

  > succeed (+) |> take intDigit
    <function> : Parser.Parser (Int -> Int)

  > succeed (+) |> take intDigit |> take intDigit
    <function> : Parser.Parser Int

  > run (succeed (+) |> take intDigit |> take intDigit) "12"
  Just 3 : Maybe.Maybe Int
  ```

- or with `(,)`:

  ```
  mkTuple = succeed (,)
      |> take digit
      |> take digit
  ```

  ```
  > run mkTuple "50"
    Just ('5','0') : Maybe ( Char, Char )
  ```

Ignore stuff

`drop` is a parser that applies another parser (which must match) but ignores its output.

```
drop : Parser drop -> Parser keep -> Parser keep
drop dropper keeper =
    keeper
        |> andThen
            (\value ->
                dropper
                    |> andThen (\_ -> succeed value)
            )
```

- It works like this:

  ```
  > (upper |> drop digit ) "A5"
    Just ('A',"") : Maybe.Maybe ( Char, String )
  ```

  Which seems odd, but wait!

Now we can chose what to `take`, and what to `drop`:

```
pair : Parser ( Int, Int )
pair =
    succeed (,)
        |> drop (char '(')
        |> take intDigit
        |> drop (char ',')
        |> take intDigit
        |> drop (char ')')
```

- Discard concrete syntax; keep concrete syntax:

  ```
  > run pairN "(4,3)"
    Just (4,3) : Maybe.Maybe ( Int, Int )
  ```

Parser pipelines

`elm-tools/Parser` does this with its own infix pipe operators:

```
apply : (a -> b) -> a -> b
apply f a =
    f a

map2 : (a -> b -> value) -> Parser a -> Parser b -> Parser value
map2 f pa pb =
    pa |> andThen (\a -> pb |> map (\b -> f a b))

-- This is take
infixl 5 |=
(|=) : Parser (a -> b) -> Parser a -> Parser b
(|=) pf pa =
    map2 apply pf pa

-- this is drop
infixl 5 |.
(|.) : Parser keep -> Parser ignore -> Parser keep
(|.) keeper ignorer =
    map2 always keeper ignorer
```

So we can write

```
pair1 : Parser ( Int, Int )
pair1 =
    succeed (,)
        |. char '('
        |= num
        |. char ','
        |= num
        |. char ')'
```

```
> run pair1 "(5,0)"
  Just (5,0) : Maybe ( Int, Int )
```

Repetition

Suppose we want to take several digits, e.g. a list of characters:

- can use cons `(::)`

  ```
  > succeed (::)
    <function> : Parser.Parser (a -> List a -> List a)

  > succeed (::) |> take digit
    <function> : Parser.Parser (List Char -> List Char)
  ```

Now we need a parser `digits : Parser (List Char)`

`succeed (::) |>; take digit |>; take digits`

- We would write this recursively: **either** the input has some number of digits, in which case we **take** the first and cons it with the rest **or** no digits -- our list is empty

  ```
  {--
    digits : Parser (List Char)
    digits =
      either
          (succeed (::)
              |> take ditig
              |> take digits
          )
          (succeed [])
  --}
  ```

Either

If the first parser succeeds, we're done, otherwise use the second one

```
either : Parser a -> Parser a -> Parser a
either p1 p2 =
    \inp ->
        case p1 inp of
            Just result ->
                Just result

            Nothing ->
                p2 inp
```

- More consisely, using Maybe:

```
either : Parser a -> Parser a -> Parser a
either parser1 parser2 =
    \inp ->
        parser1 inp
        |> Maybe.map Just
        |> Maybe.withDefault (parser2 inp)
```

Many digits

```
digits : Parser (List Char)
digits =
    either
        (succeed (::)
            |> take ditig
            |> take digits -- Uh oh!
        )
        (succeed [])
```

`digits` is defined in terms of itself and (eager) Elm doesn't like it. Force it to be lazy: hide the recursion into an anonymous function, using `lazy`:

```
lazy : (() -> a -> b) -> (a -> b)
lazy f =
    \a -> f () a
```

```
digits : Parser (List Char)
digits =
    either
        (succeed (::)
            |> take digit
            |> take (lazy (\() -> digits))
        )
        (succeed [])
```

- now we can parse as many digits as we can find:

  ```
  > run digits "123abc"
    Just ['1','2','3'] : Maybe.Maybe (List Char)
  ```

Many

Generalising:

- `digit` becomes any parser `p : Parser a`

```
many : Parser a -> Parser (List a)
many parser =
    either
        (succeed (::)
            |> take parser
            |> take (lazy (\() -> many parser))
        )
        (succeed [])
```

- Now we can rewrite `digits`:

```
digits0 : Parser (List Char)
digits0 =
    many digit
```

- and run it:

```
> run digits "123abc"
  Just ['1','2','3'] : Maybe.Maybe (List Char)
```

Mutli-digit integers

`many intDigit : Parser.Parser (List Int)` so we just need a way to convert `List Int` to `Int`

```
intFromList : List Int -> Int
intFromList =
    List.foldl (\i a -> a * 10 + i) 0
```

```
integer : Parser Int
integer =
    many intDigit
        |> map intFromList
```

Detecting the end of input

```
end : Parser ()
end =
    \inp ->
        if String.isEmpty inp then
            Just ( (), inp )
        else
            Nothing
```

And ignore the unit `()` with `drop`:

```
justA : Parser Char
justA =
    char 'A'
        |> drop end
```

```
> run justA "a"
Nothing : Maybe Char

> run justA "A"
Just 'A' : Maybe Char

> run justA "AA"
Nothing : Maybe Char
```