

Demonstrating the other elementary parser:

```
ParseLib> papply mzero "anything"
ERROR: Cannot find "show" function for:
*** Expression : papply mzero "anything"
*** Of type    : [(a,String)]
```

Oops! This is one of those cases where there's no context to determine the type of the "delivered item", and so Hugs can't determine how to display it.

So we declare a type:

```
ParseLib> (papply mzero "anything") :: [(Bool,String)]
[]
```

(Since the list is empty, it doesn't matter which "showable" type we supply.)

More demonstrations:

```
ParseLib> papply (return 3) "a string"
[(3,"a string")] --Note that the entire string is "remaining".
```

```
ParseLib> papply (sat isDigit) "12345"
[('1',"2345")]
```

```
ParseLib> papply (sat isLower) "12345"
[]
```

Function `sat` can be used to define many simple but useful parsers:

```
> digit :: Parser Char
> digit = sat isDigit
> lower :: Parser Char
> lower = sat isLower
> upper :: Parser Char
> upper = sat isUpper
> letter :: Parser Char
> letter = sat isAlpha
> alphanum :: Parser Char
> alphanum = sat isAlphaNum
```

and to define a function that returns a parser which recognizes a given character:

```
> char :: Char -> Parser Char
> char c = sat (\y -> c == y)
ParseLib> papply (char 'x') "xyyzz"
[('x',"yyzz")]
ParseLib> papply (char 'x') "zzyyxx"
[]
```

Like IO actions, Parsers can be combined to make "bigger" parsers.

There are two fundamental ways in which Parsers can be combined:

- **sequentially**, using `do` notation
- **alternatively**, using operator `(+++)`.

Sequential combination

If `p1` and `p2` are parsers, then using their *sequential* combination

```
do { p1; p2 }
```

to parse a string

- first uses `p1` to parse the original string
- then uses `p2` to parse the remaining string

A demonstration:

```
ParseLib> papply ( do { digit; lower } ) "3ab"
[('3',"ab")]
```

If *either* of the sequentially combined parsers fails, the combination fails:

```
ParseLib> papply ( do { digit; lower } ) "aab"
[]
```

```
ParseLib> papply ( do { digit; lower } ) "33b"
[]
```

```
ParseLib> papply ( do { digit; lower } ) "a3b"
[]
```

The values delivered by parsers in a `do` can be named:

```
let get2 = do x <- item
             y <- item
             return (x,y)
```

```
in papply get2 "4bcd"
```

```
--> [[('4','b'),"cd"]]
```

Sequential combination is what is used to define the parser `sat` p:

```
sat :: (Char->Bool) -> Parser Char
sat p = do x <- item
         if p x then return x else mzero
```

Recall from our expression grammar (slide 29):

```
Factor ::= '(' Expression ')'
```

If `expression` were a parser for *Expression*, we could define a parser for this kind of *Factor* by

```
factor :: Parser Expression
factor = do char '('
           exp <- expression
           char ')
           return exp
```

Defining `Parser` as a monadic type enables parsers to be combined in sequence with no explicit reference to the string that's passed from each parser to its successor.

Alternative combination

If `p1` and `p2` are parsers, then parsing a string using their *alternative* combination `p1 +++ p2`

- uses `p1` to parse the string
- if `p1` *succeeds*, delivers `p1`'s result
- if `p1` *fails*, uses `p2` to parse the string
- if `p2` *succeeds*, delivers `p2`'s result
- if `p2` *fails*, the combination `p1 +++ p2` fails.

Examples:

```
ParseLib> papply digit "12ab"
[('1',"2ab")]
```

```
ParseLib> papply letter "12ab"
[]
```

```
ParseLib> papply digit "ab12"
[]
```

```
ParseLib> papply letter "ab12"
[('a',"b12")]
```

```
ParseLib> papply (digit +++ letter) "12ab"
[('1',"2ab")]
```

```
ParseLib> papply (digit +++ letter) "ab12"
[('a',"b12")]
```

```
ParseLib> papply (digit +++ letter) "... "
[]
```

The production for *Factor* actually includes three alternatives:

```
Factor ::= '( Expression )' | Name | Numeral
```

These can be handled by defining `factor` as an alternative combination of three parsers:

```
> factor :: Parser Expression
> factor = do char '('
>           exp <- expression
>           char ')'
>           return exp
>
>           +++ name
>           +++ numeral
```

Repetition

The function

```
many :: Parser a -> Parser [a]
```

is defined so that

```
many p
```

is a parser which

- recognizes *zero or more* occurrences of what `p` recognizes
- delivers the items delivered by `p` as elements of a list

Example: decimal numerals

```
ParseLib> papply (many digit) "1234...."
[("1234","....")]
```

Delivering a numeral's `Int` value

```
> nat :: Parser Int
> nat = do s <- many digit
>       return (read s)
```

```
ParseExamples> papply nat "1234...."
[(1234,"....")] -- 1234 is Int, not String
```

```
ParseExamples> papply nat "... "
Program error: Prelude.read: no parse
```

A numeral should contain at least one digit, but `many digit` accepts empty digit strings.

So we modify `nat` to require at least one digit:

```
> nat :: Parser Int
> nat = do d <- digit
>         ds <- many digit
>         return (read (d:ds))
```

```
ParseExamples> papply nat "... "
[]
```

```
ParseExamples> papply nat "54321xyz"
[(54321,"xyz")]
```

A parser that recognizes *signed* numerals

```
> num :: Parser Int
> num = do char '-'; n <- nat; return (-n)
>
>         +++ nat
```

Using parsers

A successful parse of a string

- delivers a result
- "uses up" the string

A typical way to use a parser in a larger program:

```
> case papply parser string of
>   [(result,"")] -> do something with result
>   _              -> ... parser failed ...
```