opt-parse - An Advanced Command Line Parser for Curry

opt-parse is an advanced command line parser for Curry. It features support for options with and without values (i.e. flags), positional arguments and commands that can define their own sub-parsers. It borrows heavily from Paolo Capriotti's Haskell package optparse-applicative and Curry's GetOpt module.

You use opt-parse by declaring a *parser specification* and then running that parser specification on a command line. A parser specification is made up from individual parsers for options, flags, position arguments and commands. Each individual parser results in an arbitrary value, though all parsers in a parser specification must result in values of the same type.

A Simple Example

A simple command line parser example might look like this:

```
cmdParser = optParser $
     option (\s -> readInt s)
       (  long "number"
       <> short "n"
       <> metavar "NUMBER"
       <> help "The number." )
 <.> arg (\s -> readInt s)
       (  metavar "NEXT-NUMBER"
       <> help "The next number." )

main = do
  args <- getArgs
  parseResult <- return $ parse (intercalate " " args) cmdParser "test"
  putStrLn $ case parseResult of
    Left err -> err
    Right  v -> show v
```

This defines a parser that supports a `number` option and requires a single positional argument. Both values are parsed into an integer. The `parse` function is called with the command line as a single string, the parser specification and the name of the current program. It results in either a `Left` if there was a parse error or a `Right` with the list of parse results. Running `test --help` prints out usage information:

```
test NEXT-NUMBER
```

```
-n, --number NUMBER      The number.

 NEXT-NUMBER     The next number.
```

If we run `test --number=5 2`, we get the list of parse results:

```
[2, 5]
```

`metavar` and `help` are modifiers that can be applied to any argument parser, command, option, flag or positional. The `help` text is what is printed in the detailed usage output, the `metavar` is the placeholder to be printed for the argument's value in the usage output. The `optional` modifier can also be applied to all argument types, although flags and options are already optional by default.

The `long` and `short` modifiers are specific to options and flags.

Right now, the result of our parser is a list of the individual parse results. Usually, we want our parse result to be a single value, for example a Curry data type such as this:

```
data Options = Options
  { number :: Int
  , nextNumber :: Int }
```

To parse a command line to an `Options` value, we return functions from our individual parsers instead of integers:

```
cmdParser = optParser $
     option (\s a -> a { number = readInt s })
       (  long "number"
       <> short "n"
       <> metavar "NUMBER"
       <> help "The number." )
 <.> arg (\s a -> a { nextNumber = readInt s })
       (  metavar "NEXT-NUMBER"
       <> help "The next number." )
```

The result of a successful parse will now be a list of functions that change an `Options` value. We can fold this list onto a default `Options`:

```
applyParse :: [Options -> Options] -> Options
applyParse fs = foldl (flip apply) defaultOpts fs
 where
  defaultOpts = Options 0 0
```

```
main = do
  args <- getArgs
  parseResult <- return $ parse (intercalate " " args) cmdParser "test"
  putStrLn $ case parseResult of
    Left err -> err
    Right  v -> show $ applyParse v
```

Executing `test --number=5 1` results in:

```
(Options 5 1)
```

Positional Arguments and Flags

Positional arguments can be created via `arg` and `rest`. `arg` is a normal positional argument which can be optional or mandatory. `rest` is a positional argument that consumes the rest of the command line as-is. Positional arguments are expected in the order they occur in the parser definition.

`flag` can be used to create flag arguments. A flag argument expects no value.

Commands

In addition to options, flags and positional arguments, opt-parse also includes support for commands. A command is a positional argument that dispatches to sub-parsers depending on its value. If we have a calculator program that supports addition and multiplication, we could model its command line interface using commands:

```
data Options = Options
  { operation :: Int -> Int -> Int
  , operandA :: Int
  , operandB :: Int }

cmdParser = optParser $
  commands (metavar "OPERATION")
    (   command "add" (help "Adds two numbers.") (\a -> a { operation = (+) })
          (   arg (\s a -> a { operandA = readInt s }
                (  metavar "OPERAND-A"
                <> help "The first operand." )
          <.> arg (\s a -> a { operandB = readInt s }
                (  metavar "OPERAND-B"
                <> help "The second operand." ) )
    <|> command "mult" (help "Multiplies two numbers.") (\a -> a { operation = (*) })
```

```
(   arg (\s a -> a { operandA = readInt s }
      (  metavar "OPERAND-A"
      <> help "The first operand." )
<.> arg (\s a -> a { operandB = readInt s }
      (  metavar "OPERAND-B"
      <> help "The second operand." ) ) )
```

The corresponding usage output for `test` run with no further arguments is:

```
test OPERATION

Options for OPERATION
add      Adds two numbers.
mult     Multiplies two numbers.
```

If we choose an operation, e.g. `add`, the output is:

```
test add OPERAND-A OPERAND-B

OPERAND-A     The first operand.
OPERAND-B     The second operand.
```