SIMPLE

Daniel Zhu
Compilers & Interpreters

05 January 2026

Abstract

From scratch in Haskell, we implement both parser combinators and an interpreter for the SIMPLE language.

1 Grammar

We represent the grammar of SIMPLE in Haskell like so:

```
type Id = String

type Program = [Statement]

data Statement
  = Display Expr (Maybe Id)
  | Define Id Expr
  | While Expr Program
  | If Expr Program (Maybe Program)
  deriving (Show, Eq)

data Expr
  = Binary String Expr Expr
  | Var Id
  | Num Int
  deriving (Show, Eq)
```

For instance, the program `display x + 2 read y` would be represented as

```
[Display (Binary "+" (Var "x") (Num 2)) (Just "y")]
```

2 Parsing

We opt to merge the lexing phase into the parsing phase and simply treat each individual character as a token/lexeme. A token is simulated by the `symbol` parser, which can parse keywords such as `while` or operators such as <>.

```
symbol s = try (string s <* spaces) <|> parseError s ""
```

In words, this attempts to parse a given string `s` and also any subsequent spaces, and if it fails, an error is thrown.

2.1 Parsers

We define the `Parser` type as follows:

```
newtype Parser a = Parser {runParser :: String -> (String, Either ParseError a)}
```

`Parser a` takes in a source file (`String`) and parses an AST of type `a`. It then returns this AST alongside the remaining contents of the file.

For instance:

```
pVar :: Parser Var
-- parse an identifier
pVar = ...

print $ runParser pVar "x = 5"
```

should output:

```
("= 5", Right (Var "x"))
```

2.2 Combinators

One of the most elegant ways to construct parsers is to build them up via *parser combinators*. Some examples:

- **Choice (`<|>`)**: Let's say we want a parser that recognizes either the string "red" or the string "blue". If we had `pRed` and `pBlue` to recognize red and blue individually, we could write our desired parser as `pRed <|> pBlue` using the choice (`<|>`) combinator.
- **Sequence (>>)**: Or, if we wanted to parse "redblue", we could write `pRed >> pBlue` using the sequence (>>) combinator.
- **Many (`many`)**: If we wanted to parse the string "redredredred…red", we could write `many pRed`.

With just these three combinators, we can build all the way up to the entirety of the SIMPLE language. For instance, we can write our top level program/statement parsers like so:

```
pProgram :: Parser Program
-- remove whitespace from the start
pProgram = spaces *> many pStatement

pStatement :: Parser Statement
pStatement = choice "statement" [pDisplay, pDefine, pWhile, pIf]
```

where `choice` is shorthand for multiple parsers connected by `<|>`. For more implementation details, please consult Section 6.

2.3 Backtracking

We enable parsers to backtrack via the `try` function:

```
try :: Parser a -> Parser a
try p = Parser $ \s -> case runParser p s of
  (_s', Left err) -> (s, Left err)
  success -> success
```

It takes in a parser p, and when that parser fails, reverts s to its original state.

Allowing every possible choice to backtrack is clearly inefficient, so an important part of our design is deciding when to allow backtracking. Consider parsing a Statement, which can begin with any of the four keywords display, define, while, or if. We want to allow pDisplay, which is responsible for parsing Display statements, to backtrack if the first token is not display. However, if the first token is display, we should not allow backtracking afterward, since there is no other statement type that begins with this keyword. This results in the following structure:

```
pStatement :: Parser Statement
pStatement = choice "statement" [pDisplay, pDefine, pWhile, pIf]

pDisplay :: Parser Statement
pDisplay = do
  try $ symbol "display"
  -- ...

pDefine :: Parser Statement
pDefine = do
  try $ symbol "define"
  -- ...

pWhile :: Parser Statement
pWhile = do
  try $ symbol "while"
  -- ...

pIf :: Parser Statement
pIf = do
  try $ symbol "if"
  -- ...
```

2.4 Expression parsing

The expression parser is very customizable: we just need to provide an operations array.

```
operations = [
  ["and", "or"],
  ["<", ">", ">=", "<=", "<>", "="],
  ["+", "-"], ["*", "/"]
]
```

The operators are grouped by precedence level. Within each group, operators are *left-associative*: that is, $5 - 3 + 2 = (5 - 3) + 2 = 4$.

To parse, we define a helper function pLevel ops which parses expressions that use only the operations allowed in ops. Then, we could define something to the following effect:

```
pLevel ops@(cur : nx) = do
  lhs <- pLevel nx
  res <- maybe $ do
    op <- choice (concat sortedOps) (map (try . symbol) sortedOps)
    rhs <- pLevel ops
    return (Binary op lhs rhs)
  case res of
    Just res -> return res
    Nothing -> return lhs
```

However, note that this although this is probably the most natural way to recursively define expressions, it results in expressions which are right rather than left associative.

One way to fix this is to accumulate all the expressions at level `nx` in an array first, then combine them left-associatively afterward. For instance, for the expression $3 * 5 + 6/2 - 4 * 7$, we would first convert it to `[Binary * 3 5, (+, Binary / 6 2), (-, Binary * 4 7)]`, then into its ultimate left-associative form. This results in the following definition of `pLevel`:

```
pLevel (cur : nx) = do
  lhs <- pLevel nx
  let pNx = do
        -- sort in desending order of length e.g. so <> is tried before <
        let sortedOps = sortBy (comparing (Data.Ord.Down . length)) cur
        op <- choice (concat sortedOps) (map (try . symbol) sortedOps)
        expr <- pLevel nx
        return (op, expr)
  rem <- many pNx
  return $ foldl (\acc (op, expr) -> Binary op acc expr) lhs rem
```

Here, we use the built-in `foldl` to easily implement left-associativity.

2.5 Everything else

Implementation-wise, none of the other non-terminals are particularly noteworthy, as they all directly follow from the grammar description. For instance, this is the function to parse `While` loops:

```
pWhile :: Parser Statement
pWhile = do
  try $ symbol "while"
  cond <- pExpr
  symbol "run"
  body <- pProgram
  symbol "endwhile"
  return (While cond body)
```

We also implement a `maybe` helper function to assist with situations like parsing `else` or `read`, as these tokens may not necessarily exist:

```
pDisplay :: Parser Statement
pDisplay = do
  try $ symbol "display"
  expr <- pExpr
  id <- maybe (symbol "read" *> pId)
  return (Display expr id)
```

2.6 Error reporting

Errors will display in the format `ParseError: expected <token> at <remaining string>` like so:

```
ParseError: expected "run" at: >< trial run
    display cnt
    define ...
```

3 Interpreting

The core of our interpreter is the `Env` type, which is just a map from keys (variable names) to their corresponding values.

```haskell
newtype Env = Env {envVars :: Map String Int}
```

Then, we define the following two functions:

```haskell
evalExpr :: Env -> Expr -> Int
execStatement :: Env -> Statement -> IO Env
```

The first function evaluates an `Expr` in the context of a given `Env`, and the second evaluates a `Statement` and returns a potentially modified `Env`. We use the `IO` monad because `execStatement` can potentially produce side effects, i.e. reading input and displaying output.

3.1 Expressions

Evaluating expressions is fairly straightforward. For the two base cases of `Num` and `Var`, we just return the literal or query the `Env`, respectively:

```haskell
Num n -> n
Var x -> Map.findWithDefault 0 x env
```

For convenience, we assign a value of `0` to undefined variables.

Then, for binary operations, we simply evaluate both sides and combine them accordingly. One implementation note is that Boolean expressions will return either 1 (true) or 0 (false), so we don't have to differentiate types between `Int` and `Bool`.

3.2 Statements

Evaluating statements is also fairly straightforward. One particularly illustrative example for the monadic pattern is the implementation of `Display`/`Read`:

```haskell
Display expr readId -> do
  let val = evalExpr env expr
  print val
  case readId of
    Nothing -> return env
    Just name -> do
      putStr (name ++ ": ")
      input <- getLine
      case reads input of
        [(n, "")] ->
          let Env vars = env
           in return $ Env (Map.insert name n vars)
        _ -> ioError (userError "Invalid input, expected an integer.")
```

It reads basically as an imperative description of what `Display/Read` actually does.

4 Testing

4 test files are included in this project: `binary.simple`, `fib.simple`, and `max.simple`, and `err.simple`. There are comments in each file describing the expected behavior of each program, which will also be outlined below. To run any of them (or your own code), simply run the command `runghc main.hs <filename>`. Of course, you will have to install Haskell first.

4.1 `binary.simple`

This program takes an integer as input and outputs its binary representation.

4.2 `fib.simple`

This program takes an integer `n` as input and outputs the first `n` terms of the Fibonacci sequence.

4.3 `max.simple`

This program first asks for a number of trials, then for a pair of integers in each trial. For each pair, the program will output the maximum among the two.

4.4 `err.simple`

Self-explanatory. This code is meant to demonstrate the error-reporting capabilities of the parser.

5 Source Code

You can find the code for this project on GitHub or attached as a `.zip` file to the Schoology submission.

6 Appendix

6.1 Monads

Monads are how we represent sequential computation in Haskell. If the type `m` is a Monad, the type `m a` represents an `m`-type computation that returns a value of type `a` to be used in subsequent computations. Moreover, `m`-type computations should be able to be chained with one another to form another `m`-type computation.

For instance, the `Program` type is a Monad because two `Program`s can be chained together to form another `Program`. In Haskell, a general `Program` is represented by the `IO` monad. Two `Parsers` in sequence form another parser, so the `Parser` type is also a Monad.

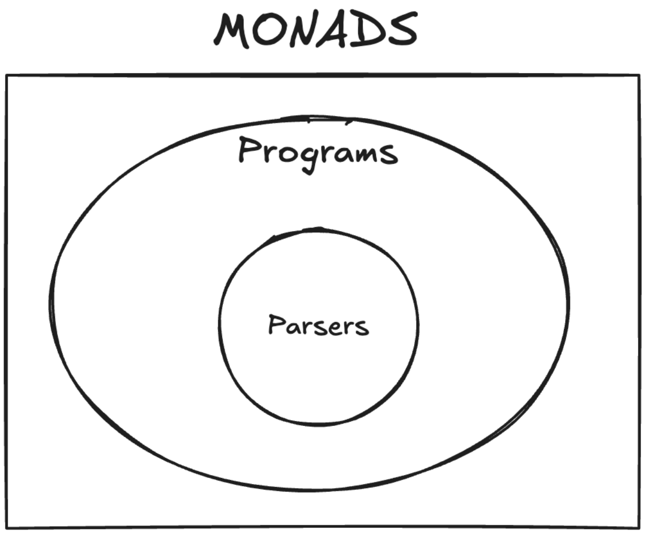

Figure 1: A diagram showing the hierarchy of `Monad`, `Program`, and `Parser`.

The chaining of two `Parsers` is described as follows:

```haskell
instance Monad Parser where
  pa >>= f = Parser $ \s -> case runParser pa s of
    (s', Right a) -> runParser (f a) s'
    (s', Left e) -> (s', Left e)
```

If we encounter an error, we stop execution; otherwise, we pass the remaining string on to the next `Parser`. The advantage of defining chaining in this way is that we no longer need to do any explicit error handling: if anything goes wrong, the `Monad` will catch it automatically.