Session 9

Case study: a parsing library

This session

- *Parsing* is the process of extracting structured data from strings in an agreed format, e.g. `"(x + y)* 5.6"`.
- A common approach in functional languages is to use a library supplying operators that enable one to write parsers reflecting a context-free grammar. (recursive descent parsers)
- We shall illustrate with a simple library of this sort.

Context-free grammars

Example task

Our first task is parsing code in a simple assembly language, *e.g.*

```
cpy 1 a
cpy 1 b
cpy 26 d
jnz c 2
jnz 1 5
cpy 7 c
inc d
dec c
jnz c -2
```

1. define data types to represent code
2. give a context-free grammar for instructions
3. use the library to implement it

Representing instructions

The type definitions mirror the grammar:

```haskell
type Code = [Instruction]

data Instruction
    = Copy Value Reg
    | Incr Reg
    | Decr Reg
    | JNZ Value Int
    deriving Show

data Value = Register Reg | Constant Int
    deriving Show

type Reg = Char
```

A context-free grammar for instructions

We can use `lines` to break the input into strings containing instructions, so it remains to analyse

instruction → `"cpy"` *space value space reg*
| `"inc"` *space reg*
| `"dec"` *space reg*
| `"jnz"` *space value space int*

those.

value → *reg*
| *int*

reg → (lowercase letter)

An *attribute grammar* is a context-free grammar in which each symbol has an associated value, with rules for computing the value of a symbol on the left side of a production from those on the right size.

Attribute grammar for instructions

In our example, the nonterminals *instruction*, *value* and *reg* will have attributes of type `Instruction`, `Value` and `Reg` respectively, and the attribute rule for each production can be expressed as a function:

instruction → `"cpy"` *space value space reg*
`\ _ _ v _ r -> Copy v r`
| `"inc"` *space reg*
`\ _ _ r -> Incr r`
| `"dec"` *space reg*
`\ _ _ r -> Decr r`
| `"jnz"` *space value space int*
`\ _ _ v _ n -> JNZ v n`

We ignore the values associated with some symbols.

Parsers

The `Parser` library

The **Parser** library defines a type **Parser a** of parsers that scan the front part of an input string and convert it to a value of type **a**.

We will define parsers for each nonterminal of the grammar:

```
instruction :: Parser Instruction
value :: Parser Value
reg :: Parser Reg
```

The library provides primitive parsers for literal characters and strings, as well as operators including

p1 <*> p2: match the first part of the string with **p1**, and then match (part of) the rest with **p2** (concatenation)

p1 <|> p2: parse the string with either **p1** or **p2** (choice)

which we can use to provide definitions of our parsers that mirror the context-free grammar.

Recursive descent parsing

Our final parser (**instruction**) will be matched against a whole string, using the function

Parser
```
parseAll :: Parser a -> String -> [a]
```

Ideally, parsing a string should return exactly one value.

- It will return no value (an empty list) if the parse fails.
- It may return more than one if the grammar is ambiguous.

The **Parser** type

How should **Parser a** be represented?

- First guess: **String -> a**
- To glue parsers together to make new parsers, we need to parse the front part of the input string and return the rest: **String -> (a, String)**
- We can allow for failure and local ambiguity by returning a list of possible parses: **String -> [(a, String)]**
- Wrapping this in a **data** type:

  Parser
  ```
  data Parser a = P (String -> [(a, String)])
  ```

- Parsing a prefix of a string:

  Parser
  ```
  parsePrefix :: Parser a -> String -> [(a, String)]
  parsePrefix (P p) = p
  ```

Other representations are possible.

Matching single characters

This primitive parser matches any character for which the predicate returns `True`:

Parser
```
satisfy :: (Char -> Bool) -> Parser Char
```

Special cases are also provided:

Parser
```
space :: Parser Char
space = satisfy isSpace
```

and similarly `digit` and `letter`.

interpreter
```
parsePrefix space " hello"
parsePrefix space "hello"
parsePrefix letter "abc"
parsePrefix digit "r2d2"
parsePrefix digit "2d2"
```

Parsing register names and integers

We can now define a parser for a register (a lowercase letter):

```
reg :: Parser Reg
reg = satisfy isLower
```

interpreter
```
parsePrefix reg " b c"
parsePrefix reg "a b"
```

The library supplies a primitive parser for numbers:

Parser
```
int :: Parser Int
```

interpreter
```
parsePrefix int "123 rest"
parsePrefix int "123"
parseAll int "123"
```

Parsing values

Recall the grammar for values:

$$\begin{array}{lcl} \textit{value} & \rightarrow & \textit{reg} \\ & | & \textit{int} \end{array}$$

We have defined `reg :: Parser Reg`, and the library defines `int :: Parser Int`.

Recall the data type definition

```
data Value = Register Reg | Constant Int
```

The library has a "map-like" operator applying a function to values returned by a parser:

Parser
```
(<$>) :: (a -> b) -> Parser a -> Parser b
```

with which we can turn these into parsers returning values:

```
Register <$> reg :: Parser Value
```

```
Constant <$> int :: Parser Value
```

Choice between parsers

The library has an operator for combining alternative parsers:

Parser
```
(<|>) :: Parser a -> Parser a -> Parser a
```

We can use this to combine our two parsers to get a parser for values:

```
value :: Parser Value
value = Register <$> reg <|> Constant <$> int
```

Testing:

interpreter
```
parsePrefix value "b more stuff"
parsePrefix value "123 more stuff"
parseAll value "b"
parseAll value "123"
```

Literal matching

The library has a primitive making a parser that matches a given character:

Parser
```
char :: Char -> Parser Char
char c = satisfy (== c)
```

and also one that matches a specified string:

Parser
```
string :: String -> Parser String
```

Both of these parsers yield values, but they are not very interesting, because they are the same as the values we specified.

interpreter
```
parseAll (char 'a') "a"
parseAll (char 'a') "b"
parseAll (string "abc") "abc"
```

Sequencing

We have parsers

Parser
```
letter :: Parser Char
digit :: Parser Char
```

We can apply each or these

interpreter
```
parseAll letter "r2d2"
parseAll digit "2d2"
```

Suppose we wish to run these two parsers in sequence, by running the second on the residual string from the first, and combine the `Char`s they produce as a pair:

interpreter
```
parseAll ((,) <$> letter <*> digit) "r2d2"
```

Concatenation of parsers

In general, suppose we have parsers

```
p1 :: Parser t1
```

```
p2 :: Parser t2
```

Suppose we want a parser that does the two in sequence, and then combines their results with `f :: t1 -> t2 -> t`. We have

```
f <$> p1 :: Parser (t2 -> t)
```

We can combine this with `p2` using the library operator

Parser
```
(<*>) :: Parser (a -> b) -> Parser a -> Parser b
```

so that

```
f <$> p1 <*> p2 :: Parser t
```

(The operators group to the left.)

General case

We can sequence parsers

```
p1 :: Parser t1
```

⋮

```
pn :: Parser tn
```

and combine their results with `f :: t1 -> ... tn -> t` with

```
f <$> p1 <*> ... <*> pn :: Parser t
```

which is equivalent to

```
(...(f <$> p1)<*> ...)<*> pn :: Parser t
```

Examples:

interpreter
```
parseAll ((,) <$> letter <*> digit) "r2"
parseAll ((,) <$> letter <*> int) "r2"
```

Parsing instructions

Applying this to the instruction grammar:

```
instruction :: Parser Instruction
instruction =
    (\ _ _ v _ r -> Copy v r) <$>
        string "cpy" <*> space <*> value <*> space <*> reg <|>
    (\ _ _ r -> Incr r) <$>
        string "inc" <*> space <*> reg <|>
    (\ _ _ r -> Decr r) <$>
        string "dec" <*> space <*> reg <|>
    (\ _ _ v _ n -> JNZ v n) <$>
        string "jnz" <*> space <*> value <* space <*> int
```

Ignoring values of parsers

Because parsers for literal values (like `char` and `string`) return values we do not need and wish to ignore, the library provides specialized versions of the operators that discard them:

Parser

```
(<$) :: a -> Parser b -> Parser a
x <$ p = (\ _ -> x) <$> p

(<*) :: Parser a -> Parser b -> Parser a
p <* q = (\ x y -> x) <$> p <*> q

(*>) :: Parser a -> Parser b -> Parser b
p *> q = (\ x y -> y) <$> p <*> q
```

Mnemonic: remove the angle bracket from the side whose value you wish to ignore.

interpreter

```
parseAll ((+) <$> int <* space <*> int) "123 3456"
```

Final version

Using these operators, we can simplify our parser to:

```
instruction :: Parser Instruction
instruction =
    Copy <$ string "cpy" <* space <*> value <* space <*> reg <|>
    Incr <$ string "inc" <* space <*> reg <|>
    Decr <$ string "dec" <* space <*> reg <|>
    JNZ <$ string "jnz" <* space <*> value <* space <*> int
```

Testing:

interpreter

```
parseAll instruction "cpy 1 a"
parseAll instruction "inc d"
parseAll instruction "dec c"
parseAll instruction "jnz c 2"
```

Example: leaf trees

Recall leaf trees:

```
data LTree a = Leaf a | Branch (LTree a) (LTree a)
    deriving Show
```

Parser for a leaf tree of numbers:

```
ltree :: Parser (LTree Int)
ltree =
    Leaf <$> int <|>
    Branch <$ char '<' <*> ltree <* char ',' <*> ltree <* char '>'
```

Examples:

interpreter

```haskell
module Instructions where

import Parser
import Data.Char

type Code = [Instruction]

-- instructions for a simple machine
data Instruction
    = Copy Value Reg
    | Incr Reg
    | Decr Reg
    | JNZ Value Int
    deriving Show

-- a value is either a register or a number
data Value = Register Reg | Constant Int
    deriving Show

-- registers are named by lowercase letters
type Reg = Char

-- parse a string containing several instructions
-- (misparses are silently discarded)
parseCode :: String -> Code
parseCode = concat . map (parseAll instruction) . lines

instruction :: Parser Instruction
instruction =
    Copy <$ string "cpy" <* space <*> value <* space <*> reg <|>
    Incr <$ string "inc" <* space <*> reg <|>
    Decr <$ string "dec" <* space <*> reg <|>
    JNZ <$ string "jnz" <* space <*> value <* space <*> int

value :: Parser Value
value = Register <$> reg <|> Constant <$> int

reg :: Parser Reg
reg = satisfy isLower
```

Figure 9.1: `Instruction.hs`: instruction parser

```
parseAll ltree "3"
parseAll ltree "<3,7>"
parseAll ltree "<2,<3,7>>"
```

More functions

Other useful functions

We may want to match the empty string, in which case we need to supply the value of the parser:

Parser
```
pure :: a -> Parser a
```

This function is used in these utility functions for matching lists of things:

Parser
```
some :: Parser a -> Parser [a] -- one or more
some p = (:) <$> p <*> many p

many :: Parser a -> Parser [a] -- zero or more
many p = some p <|> pure []
```

Things to try:

interpreter
```
parseAll int "12345"
parseAll (some digit) "12345"
```

Lists with separators

A common case is one more more things separated by some fixed text (*e.g.* commas or space) that we want to ignore:

Parser
```
sepBy1 :: Parser a -> Parser sep -> Parser [a]
sepBy1 p sep = (:) <$> p <*> many (sep *> p)
```

Things to try:

interpreter
```
parseAll (sepBy1 int (some space)) "123  34 67"
parseAll (sepBy1 (some digit) (some space)) "123  34 67"
```

Notes

- The grammar should not be ambiguous – each string should have a unique interpretation.
- As with other recursive descent parsers, left-recursive grammars will cause an infinite loop and must be avoided. They should be restructured, possibly using `some` or `many`.
- Left factoring improves performance, but is not essential.
- This is a very simple implementation. More efficient parsing libraries with similar interfaces are available.
- Many of the combining operators have more general types involving the `Functor`, `Applicative` and `Alternative` classes, of which `Parser` is just one instance.

Exercises

These exercises use the **`Parser`** module, which is on Moodle. You won't need to read the source, but you should refer to the documentation, also on Moodle.

1. Write a parser for the **`Command`** type of the **`Turtle`** module from sessions 3 and 5, to accept strings like:

   ```
   LEFT
   RIGHT
   FORWARD 23
   PEN UP
   PEN DOWN
   ```

2. (a) Define a parser

   ```haskell
   ident :: Parser String
   ```

   for identifiers, which consist of a letter followed by zero or more letters or digits.

   (b) Recall rose trees from last week:

   ```haskell
   data RTree a = RNode a [RTree a]
       deriving Show
   ```

   Write a parser

   ```haskell
   term :: Parser (RTree String)
   ```

   for terms of the forms

   ident

   ident (*term*, . . . , *term*)

3. A *particle system* is a simple physics engine in which all objects are particles, with no shape or dimensions and not interacting with each other. These properties make it feasible to simulate large numbers of particles. Particle systems are used for a variety of graphical effects.

   Real particle systems would have multiple properties and forces, but we will define a simplistic particle system by defining a particle as a pair of points (from the **`Geometry`** module of previous weeks), its current position and velocity:

   ```haskell
   data Particle = Particle Point Point
       deriving Show
   ```

   Then a particle system is a collection of these:

   ```haskell
   type System = [Particle]
   ```

   (a) Define a parser for particles, which should work on lines like these:

```
position=< 9,  1> velocity=< 0,  2>
position=< 7,  0> velocity=<-1,  0>
position=< 3, -2> velocity=<-1,  1>
position=< 6, 10> velocity=<-2, -1>
position=< 2, -4> velocity=< 2,  2>
position=<-6, 10> velocity=< 2, -2>
```

(b) Define a function

```
move :: Particle -> Particle
```

to more a particle one step (by adding its velocity to its position).

(c) Define a function

```
states :: System -> [System]
```

producing the infinite list of system states from a given initial state, formed by moving each particle on each step.