Advanced Programming 2024
Shallow Embedding Monads
Assignment 9

May 8 2025

1 Goal

After making this assignment you can design and implement a shallow embedded DSL. Moreover, you can use and implement the type classes `Functor`, `Applicative`, `Monad`, and `MonadPlus` for a new type. The DSL for this assignment is a parser.

Grammars are used to specify the syntactic structure of languages. We assume that the tokens in the grammar are strings and that the grammar is not left-recursive. The datatype `Gram` is a deep embedded DSL used to specify grammars. There is no arbitrary repetition operator such as the Kleene star, but there are recursive definitions to obtain the effect of repetition.

```
:: Gram
 = Lit     String       // the given string as a literal
 | Idn                  // Identifier: a String starting with an isAlpha character (see StdChar)
 | Int                  // integer denotation
 | Seq     [Gram]       // sequence of grammar elements
 | Alt     [Gram]       // choice between alternative grammar elements
 | Def Name Gram Gram   // assign the first grammar to the given name, yield second grammar
 | Var Name             // represents the grammar of this name
:: Name :== String
```

The grammar to describe named integer lists is (note the recursive `Var "list"`):

```
listIntGram :: Gram
listIntGram
  = Def "list" (Alt [ Lit "[]"
                    , Seq [ Lit "["
                          , Int
                          , Lit ":"
                          , Var "list"
                          , Lit "]"
                          ]
                    ])
    (Seq [Idn, Lit "=", Var "list"])
```

Note that definitions with `Def` can occur anywhere in the grammar, not just at the outermost level. One of the inputs described by this grammar is:

```
listIntInput = ["mylist", "=", "[", "7", ":", "[", "42", ":", "[]", "]", "]"]
```

In this assignment you develop a shallow embedded variant of this DSL that is a parser for these grammars.

2 Assignment

2.1 Monadic tooling

For a monadic parser we define:

```
:: Input :== [String]
:: Parse a = P (Input -> ?(a, Input))
```

Define instances of `Functor`, `pure`, `<*>`, `MonadPlus`, and `Monad` for `Parse`. The class `MonadPlus` contains `mzero` indicating failure and `mplus` that uses the second argument only when the first one fails. With the following **import** statements all imported classes should work out of the box.

```
import StdEnv, StdArray, Data.Maybe
import Data.Functor, Control.Applicative, Control.Monad
```

2.2 Parser Primitives

Define parsers corresponding to the primitives from the grammar: literal, identifier, integer, sequence, alternative, and definition. These should all be functions yielding a `Parse TREE`, with `TREE` defined as:

```
:: TREE
 = LIT String
 | IDN String
 | INT Int
 | SEQ [TREE]
```

2.3 Named List Parser

Define a shallow embedded version of the grammar `listIntGram`. The parse result of the input `listIntInput` should be the `TREE`:

```
SEQ [ IDN "mylist"
    , LIT "="
    , SEQ [ LIT "["
          , INT 7
          , LIT ":"
          , SEQ [ LIT "["
                , INT 42
                , LIT ":"
                , LIT "[]"
                , LIT "]"
                ]
          , LIT "]"
          ]
    ]
```

2.4 Optional: analysis

During this course we have seen two variants of a state Monad. For parsers this would be:

```
:: Parse1 a = P1 (Input -> ?(a, Input)) // used here
:: Parse2 a = P2 (Input -> (?a, Input)) // the default state Monad
```

Discuss the benefits and disadvantages of both variants. Would you prefer to use the other option for this assignment?

Deadline

To receive feedback, hand in your solution before May 14 23:59h.