B5 - Advanced Functional Programming

B-FUN-500

Bootstrap

Abstract Syntax Tree and Evaluation

1.0

Bootstrap

language: haskell, scheme
compilation: via Makefile, including re, clean and fclean rules

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

Part 0 - Install Chez-Scheme

Chez-Scheme is the reference for our interpreter, it's a good idea to have it at hand to test the conformity of our functions and interpreters.

You can install Chez-Scheme from here: `https://cisco.github.io/ChezScheme/#get`

Part 1 - Parsing

Parsing will **NOT** be covered in this bootstrap.
Parsing will **NOT** be needed for this bootstrap neither, please keep on reading.

The second bootstrap will cover in depth how to parse properly and cleanly.

Part 2 - Concrete Parsing Tree and Abstract Syntax Tree (CPT and AST)

To not spoil any CPT or AST structure, we will note < **(scheme example here)** > when you need to replace it by your own data structure, for example < **(define x 42)** >

In order to be able to evaluate your LISP program, it's important to store your input into a suitable data structure before printing it again.

We'll cut this operation in two steps : Concrete Parsing Tree and Abstract Syntax Tree

Concrete Parsing Tree (CPT)

Let's re-read the subject to find out what you really need to represent:

- Integer
- Symbols
- Lists

Knowing that, you will be able to define this kind of data structure :

```
# data Cpt = ? Int
           | ? ???
           | ? ???
           deriving Show
```

In this is a basic data structure, you'll be able to represent all that you need.
A CPT will handle all the data without putting any meaning to it, this will be done by the AST later.
A CPT represents syntax, while an AST represents semantics.

Try to adapt this format to your current needs, Integer is a given, but how would you represent a Symbols and Lists ?

Exercice 1

As said above try to fill in the Cpt data structure into something fitting to the GLaDOS project.

How would you represent each of the following Scheme code ?

```
(define x 5)
x
(if (> x 4) 1 0)
(define y (+ 5 x))
```

Exercice 2

Now let's try to use this structure to test that it's working as intended.

Write (in Haskell) the following function

```
getSymbol :: Cpt -> Maybe String
```

This function should return the symbol found in the `Cpt` in a Maybe or Nothing if the expression is not a symbol.
Don't overthink it, it shouldn't take more than a couple lines.
As a training, you can also try to write `getInteger` and `getList`.

Exercice 3

To fully finish our CPT understanding, let's write the following function

```
printTree :: Cpt -> Maybe String
```

This function will pretty print in english the content of the CPT:

```
printTree < (define x 5) >
> "a List with a Symbol 'define' followed by a Symbol 'x', a Number 5"

printTree < x >
> "a Symbol x"

printTree < (define y (+ 5 x)) >
> "a List with a Symbol 'define' followed by a Symbol 'y', (a List with a Symbol '+'
    followed by a Number 5, a Symbol 'x')"
```

The exact english phrasing is irrelevant, the point is to fully understand how to traverse the CPT.

At this point of the bootstrap, you **MAY** be able to define your language using only a CPT. However, an AST will make your life easier, so keep on reading!

Abstract Syntax Tree (AST)

Now that we've managed to represent our basic types and traverse our CPT, let's add more meaning to it by converting our tree into an Abstract Syntax Tree.

We can now give more meaning to lists to define some special forms to have the building blocks of your language.

To do that you must define a new data structure representing your AST.
In order to not get overly spread out, we'll focus on defining one special form here : define.

```
data Ast = Define ? ?
         | ? Int
         ...
         deriving Show
```

You should probably use the `record` syntax to simplify the re-reading of your code, but you do you.
We must also define a way to express pure scalar type (Integer, Symbols, Boolean…) in this AST as well.

EXERCICE 1

Complete the AST data structure above with the proper content. Feel free to use the record syntax. Repetition is key.

EXERCICE 2

It is time to verify that the AST is working properly. Write this haskell function:

```
cptToAST :: Cpt -> Maybe Ast
```

This will return an AST if the expression if recognized by your language syntax.

Once again use the Maybe monad to return Nothing if the expression is invalid.
You'll progressively add more and more expressions to your `cptToAST`.

For this exercice, handle the following examples :

```
cptToAST < 5 >
> Just (? 5)

cptToAST < symbol >
> Just (? "symbol")
```

EXERCICE 3

Now that you have handle simple case, improve your `cptToAST` by handling the `define` special form :

```
cptToAST < (define x 5) >
> Just (Define ?? ??)

cptToAST < (define x 5 6) >
> Nothing
```

EXERCICE 4

Find a way to add function application (i.e. procedure calling) to your list of special forms.
As a reminder, here is the function application syntax : `(functionName <ARG1> <ARG2> ... <ARGN>)`

With that you should be able to handle the following additions :

```
cptToAST < (+ x 4) >
> Just (Call ??)

cptToAST < (+ x (* 4 y)) >
> Just (Call [...] Call ??)

cptToAST < (define fourtyTwo (* 7 6)) >
> Just (Define ?? (Call ??))
```

Part 3 - Evaluation

Now that we've been building our AST, we can now start interpreting it, which by evaluating it!
This part could have been complicated, but because of all the work you did before with the AST, this should be fairly simple now!

Write the following function :

```
evalAST :: Ast -> Maybe Ast
evalAST < ( * (+ 4 3) 6) >
> Just (?? 42)
```

Bonus not so bonus

Read the roots of lisp : here