Principles of Programming Languages 2017W, Functional Programming

Assignment 3: Lisp Machine (16 points)

Lisp is a language based on the lambda calculus with strict execution semantics and dynamic typing. In this small project you should implement a simple Lisp machine in Haskell consisting of a parser and an interpreter.

Task 1: Lisp data

In Lisp, data as well as programs are represented as lists. Besides lists, your Lisp implementation should support the following primitive types: integers, Booleans, arbitrary string tokens, and the single value `"nil"` representing the empty list.

External data representation:

Thus, the external textual representation of Lisp data has to conform to the following grammar rules:

```
LispData = "nil" | Integer | Boolean | String | List .
Integer = ["+" | "-"] Digits
Boolean = "true" | "false" .
String = .. any word token not being an integer, Boolean or nil ..
List = "(" LispData { LispData } ")" .
```

Examples of Lisp data are:

```
nil
+123  -123   123
true  false
x   anyString  1st ..
(1 2 3)
(quote a)
(quote (a b c))
(if (= x 1) 1 (* x (fac (- x 1))))
(let ((fac (lambda (x) (if (= x 1) 1 (* x (fac (- x 1)))))))) (fac 6))
```

Internal data representation:

For representing Lisp data in your Haskell program, the following algebraic data type is provided:

```
data LispData =
      Nil
    | I Int
    | B Bool
    | S String
    | Pair {first :: LispData, second :: LispData }
    | Lambda { params :: [String], body :: LispData }
    deriving (Eq)
```

It comprises variants for

- a `Nil` value representing an empty list
- integer values (`I Int`)
- Boolean values (`B Bool`)
- String values (`S String`)
- `Pair`s with a first and a second part which both can be arbitrary data elements.
- `Lambda` values for representing function objects with formal parameter names and a body. `Lambda` values are created by evaluating lambda expressions (see below).

Representing lists:

A list is represented by a recursive structure of `Pair` data objects. That means, the first element of a `Pair` contains the first element of the list and the second element of the `Pair` points to the rest of the list. In the last `Pair` of a list representation, the second element contains the value `Nil`. For example, the list

```
(1 2 3)
```

is represented by the following `LispData` structure

```
(Pair (I 1) (Pair (I 2) (Pair (I 3) Nil)))
```

Implementing a parser for Lisp data:

Above definitions are provided in the download. Your first task is to write a parser for reading an external textual representation of Lisp data and building an internal representation in the form of `LispData` elements. The parser should be implemented based on the provided parser library.

Task 2: Interpreter

In Lisp, programs are represented as data elements, especially lists, which conform to specific rules. Your task is to write an interpreter in the form of a function `eval` which is able to evaluate a Lisp expression. Note, that `eval` also checks for the correct structure of evaluated elements and correct data types, thus implements a form of dynamic typing.

Lisp expressions:

Valid Lisp expressions are `LispData` elements which conform to the following rules:

```
LispExpr =
    "nil" |                                          -- value nil
    Bool |                                           -- Boolean values
    Integer |                                        -- integer numbers
    String |                                         -- variables
    "(" "quote" LispData ")" |                       -- protecting from evaluation
    "(" "=" LispExpr LispExpr ")" |                  -- equal operation
    "(" ArithOp LispExpr LispExpr ")" |              -- arithmetic operation
    "(" RelOp LispExpr LispExpr ")" |                -- relational operation
    "(" BoolOp LispExpr LispExpr ")" |               -- Boolean operation
    "(" "not" LispExpr ")" |                         -- not operation
    "(" "cons" LispExpr LispExpr ")" |               -- cons operation
    "(" ("car" | "cdr") LispExpr ")" |               -- car and cdr operations
    "(" "if" LispExpr LispExpr LispExpr ")" |        -- if expression
    "(" "lambda" "(" { String } ")" LispData ")" |   -- lambda function
    "(" LispExpr { LispExpr } ")" |                  -- function application
    "(" "let" "(" { Local } ")" LispData ")" .       -- let with local bindings


ArithOp = "+" | "-" | "*" | "/" | "%" .
RelOp = "<" | ">"| "<=" | ">=" .
BoolOp = "&&" | "||" .

Local = "(" String LispExpr ")" .
```

An example of a valid Lisp program is given by the following code, which first defines two locals, variables `FIVE` and `fac` in a let–the value of the latter being a function–and then computes `(fac FIVE)`:

```
(let
    (
      (FIVE 5)
      (fac (lambda (x)
               (if (= x 1)
                   1
```

```
                    (* x (fac (- x 1))))))
    )
    (fac FIVE)
)
```

Function eval:

Write a function `eval`

```
eval :: [Binding] -> LispData -> Result LispData
```

which takes a `LispData` element, additionally a list of bindings of local variables to `LispData` values, and tries to evaluate the element, possibly given a result or a failure, where a `Binding` is a pair of variable name and value

```
type Binding = (String, LispData)
```

and `Result` is a data type for representing success or failure of an evaluation

```
data Result a = Success { value :: a } | Failure { msg :: String }
     deriving (Eq, Show)
```

Evaluation rules:

Lisp expression are evaluated based on the following rules (which might result in a `Success` with a result value or a `Failure` with a message):

Primitive values: Boolean, number and `"nil"` values always evaluate to themselves with success.

Variables: A string is interpreted as a variable and, when found in the bindings, evaluates with success to the value bound to this variable. A `Failure` is returned if no binding exists.

Quote: A quote expression `"(" "quote" LispData ")"` evaluates to the `LispData` element. That means, quote protects from evaluation and the `LispData` element itself is not further evaluated but stays as-it-is. For example, a list `(quote (+ 1 2))` evaluates to list `(+ 1 2)` whereas a list `(+ 1 2)` without quote is evaluated as a plus operation and returns `3`.

Operations: The arithmetic, equal, relational, Boolean, and list expressions are evaluated by first evaluated the operand expressions (*strict evaluation*). If the evaluation of operand expressions succeed and return the required results, the operation is carried out and the result is returned as a success. If one evaluation fails or does not return the required result, the evaluation fails with a `Failure` value. Concretely, the different operations have to be evaluated as follows:

- *equal*: In an equal operation `"(" "=" LispExpr LispExpr ")"` the results of the evaluations of the two operand expressions are compared for equality and the Boolean value is returned.
- *Arithmetic operations*: The two operand expressions have to evaluate to integer values, otherwise a failure is returned. The respective integer operation ((+), (*), …) is carried out on the operand values and the result is returned.
- *Relational operations*: The two operand expressions have to evaluate to integer values, otherwise a failure is returned. The respective relational operation ((<), (>), …) is carried out on the operand values and the Boolean result is returned.
- *Logical operations, incl. not*: The operand expressions have to evaluate to Bool values, otherwise a failure is returned. The respective Boolean operation ((&&), (||), not) is carried out on the Boolean results and Boolean result is returned.
- *Cons operation*: In a cons operation `"(" "cons" LispExpr LispExpr ")"` a `Pair` data element is created with the two operand values becoming first and second element of the pair.
- *Car and cdr operations*: The operand expression has to evaluate to a `Pair` value, otherwise a failure is returned. For a `car` operation, the first element of the pair is returned, for a `cdr` operation the second element.

If: An if expression `"(" "if" LispExpr LispExpr LispExpr ")"` is evaluated by first evaluating the first expression, which has to evaluate to a Boolean. If this value is true, the second

expression is evaluated and provides the result, otherwise the third expression is evaluated. If the condition does not evaluate to a Boolean value, a `Failure` is returned.

Lambda: A lambda expression `"(" "lambda" "(" { String } ")" LispData ")"` should evaluate to a lambda value `Lambda [String] LispData`.

Function application: In a function application `"(" LispExpr { LispExpr } ")"` the first expression represents the function and therefore has to evaluate to a `Lambda` value. Then the operand expressions are evaluated (*strict evaluation*). Then the function application is performed by first establishing bindings of the formal parameter names in the lambda to the values returned by evaluating the operand expressions, and then trying to evaluate the body of the lambda with those additional bindings in place. A `Failure` value has to be returned when the first expression does not evaluate to a `Lambda` value or when the number of parameter names in the lambda does not correspond to the number or actual arguments in the function application.

Let: A let expression `"(" "let" "(" { Local } ")" LispExpr ")"` consists of local variable definitions and a body expression, where `Local` is a list `"(" String LispExpr ")"` of a variable name and an expression. A let is therefore evaluated by first establishing the local bindings of variables to values and then executing the body with the current bindings extended with these new local bindings in place. A binding is created by first evaluating the `LispExpr` and then binding the variable to the resulting value.

Others: Trying to evaluate any other Lisp data elements which does not correspond to above rules results in a failure.

Hints:

- Start with simple reduction rules and test them on simple examples.
- Use pattern matching intensively. Especially use pattern matching with `Pairs` for matching the patterns for Lisp expressions. For example the following pattern will recognize a list `(= <expr1> <expr2>)` as a valid equal operation:

```
eval bds (Pair (S "=") (Pair expr1 (Pair expr2 Nil))) = …
```

Downloads:

- For this assignment you get a partial implementation which already contains the data type definitions, some additional functions, and a main program implementing read-eval-print-loop (REPL), see below.
- Additionally a some Lisp function definitions are provided in files `fndefs1.lsp` and `fndefs2.lsp`.

Read-eval-print loop (REPL)

The read eval-print-loop (REPL) implemented in the module `Main` of the download allows evaluating Lisp expressions, e.g.,

```
LISP> (+ 1 2)
```

and also making global definitions using

```
(define <variable> <expression>)
```

for example,

```
LISP> (define x 3)
```

This global definition establishes a global binding for `x`. The variable `x` can then be evaluated

```
LISP> x
```

```
Success {value = 3}
```

and used in expressions

```
LISP> (* x 2)
Success {value = 6}
```

With `define` one can also define functions, e.g.,

```
(define sqr (lambda (x) (* x x)))
```

and then apply it

```
LISP> (sqr x)
Success {value = 9}
```

Further, with command `:load <filename>` one can load a file containing define statements, e.g.,

```
LISP> :load fndefs1.lsp
```

which will load and establish the bindings as given in the file.

Then, with command `:defs` one can print out all global bindings

```
LISP> :defs
("x",3)
("sqr",(lambda ["x"]("*" "x" "x")))
…
```

Finally, command

```
 LISP> :exit
```

will leave the REPL.