用JavaScript实现的Scheme解释器

陈仁泽 1700012774

简介

本项目用JavaScript编写了一个Scheme解释器，实现了R6RS的部分标准。

项目的运行环境为Node.js，支持REPL模式和脚本运行模式。

本项目使用到的第三方库、第三方工具有：

- PEG.js：用于生成parser。
- matches.js：用于高效编写模式匹配代码。
- yargs：用于处理命令行参数。

运行本项目不需要再额外安装其他第三方库。

使用

帮助

```
/root/of/project> node main.js --help
选项:
      --version  显示版本号                                          [布尔]
  -i, --input    input file path
      --help     显示帮助信息                                        [布尔]
```

REPL模式

在项目根目录运行：

```
/root/of/project> node main.js
$ (+ 1 2)
3
$ (define x 1)
$ x
1
$ (define l (list x (+ x 1) (+ x 2)))
$ l
(1 2 3)
$ (map (partial + 1) l)
(2 3 4)
$ (define (double x) (+ x x))
$ (map double l)
(2 4 6)
$ (define square (lambda (x) (* x x)))
$ (map square l)
(1 4 9)
$ (define (head-of h . t) h)
$ (head-of 1 2 3)
1
$ (apply head-of l)
1
$ (define tail-of (lambda (h . t) t))
$ (tail-of 1 2 3)
(2 3)
$ (apply tail-of l)
(2 3)
$ (define (fact n)
    (if (= n 0)
        1
        (* n (fact (- n 1)))))
$ (fact 10)
3628800
```

支持多行输入（键入换行符时，若已输入的字符串含有未匹配的括号，则会继续等待用户输入）。

脚本模式

假设需要运行的脚本为`/root/of/project/script.scm`：

```
(define (fib n)
  (let iter ([n n] [a 0] [b 1])
    (if (= n 0)
        a
        (iter (- n 1) b (+ a b)))))
(display (fib 10))
(newline)
```

在项目根目录运行：

```
/root/of/project> node main.js -i script.scm
55
```

测试代码

简单的测试代码位于`test.scm`中：

- 开头一小部分是对基本的相等性的测试，输出为`#t`或`#f`，预期结果用注释的方式标注在旁边。
- 之后的部分基于在该文件中定义的`check`函数，如果测试通过会输出`passed`，否则输出`failed`。

该测试文件覆盖了本解释器的大部分功能（基本语义、内置函数、库函数），可以通过该文件对本解释器的支持的功能有一个概览。

在项目根目录运行：

```
/root/of/project> node main.js -i test.scm
```

实现

类型定义

值的类型

数据类型相关的定义和操作位于`value.js`中

实现的数据类型的及其继承结构如下：

- `Value`：所有值的基类。
  - `Void`：表示空值，一般作为一些产生side-effect而不返回值的操作的返回值。
    - `Void`类型只有一个值。
  - `Nil`：表示空列表。
    - `Nil`类型只有一个值，其字面表示为`()`。
  - `Pair`：基本的组合类型，表示两个`Value`的组合，可以和`Nil`类型的值构成列表。
    - `Pair`的字面表示为`(v1 . v2)`。
    - 列表为scheme最基本的结构，字面表示形如`(v1 v2 v3)`，等价于`(v1 . (v2 . (v3 . ()))) `。
  - `Immediate`：和JS的基本类型直接对应的类型。
    - `Number`：内部为JS的number。
    - `Boolean`：内部为JS的boolean。
    - `Symbol`：内部为JS的string，用于表示scheme程序中出现的identifier。
  - `Procedure`：函数的基类。
    - `Closure`：用户定义的函数，其内部记录了函数的参数列表、函数的词法作用域以及函数体。
    - `Primitive`：解释器的内置函数，实现一些基本的操作（比如基本的代数运算）。
      - 大部分内置函数实现于`primitive.js`中。

词法作用域

词法作用域`Scope`定义于`scope.js`中，其包含的成员主要为：

- `outer`：表示外面一层的`Scope`。
- `map`：表示该层词法作用域中变量名和值的映射。

核心语义

以下列出部分核心语义（在词法作用域`scope`下对表达式`expr`的求值结果表示为`EVAL{expr | scope}`）：

- `EVAL{(define x e) | scope}` := `scope.define_value(x, EVAL{e | scope}), void`。
- `EVAL{(lambda params ...body) | scope}` := `Closure(params, body, scope)`。
- `EVAL{(set! x e) | scope}` := `scope.set_value(x, EVAL{e | scope}), void`。
- `EVAL{(Closure(params, body, scope) ...args) | scope}` :=
  `EVAL*{body | Scope(scope, Map(params => args))}`
- `EVAL*{e1 e2 ... last_e | scope}` :=
  `EVAL{e1 | scope}`, `EVAL{e2 | scope}`, ..., `EVAL{last_e | scope}`
- `EVAL{(Primitive(func) ...args) | scope}` := `func(...args)`
- `EVAL{(begin ...body) | scope}` := `EVAL*{body | scope}`
- `EVAL{(if e1 e2 e3) | scope}` :=
  `EVAL{e1 | scope} ? EVAL{e2 | scope} : EVAL{e3 | scope}`
- `EVAL{(quote e) | scope}` := `x`
- `EVAL{x | scope}` := `scope.get_value(x)`

该部分主要实现在`evaluate.js`中。

运行流程

处理命令行参数

虽然本项目需要处理的命令行参数只有`-i`/`--input`、`--help`，但还是使用了现有的轮子[yargs](yargs)（以致于安装了其他一堆依赖库……）。

读取输入

该部分功能主要实现于`io.js`中：

- `read_sexp`：读入一个完整的s-expression（可以跨多行），主要用于REPL模式的输入。
- `load_file`：读取一个文件的输入。

Parsing

scheme的语法基本还是普通的s-expression（或者说LISP语法）：

- `sexp` :=
  - ' `sexp`

    - `( sexp* )`
    - `atom`
- `atom :=`
    - `number`
    - `boolean`
    - `symbol`

Parsing部分主要就是将输入字符串转化为类似JSON的形式（不过只包括Array、String、Number和Boolean），比如`(lambda (a) (if #t a 0))`会被转化为`["lambda", ["a"], ["if", true, "a", 0]]`。

Parsing部分的实现使用到了[PEG.js](PEG.js)，这是一个基于LL文法的JS的Parser-Generator。语法描述位于`parser.pegjs`，PEG.js生成的parser位于`parser.js`。

去语法糖

主要在JSON格式的中间表示上操作，来将语法糖去除。

scheme的语法糖较为多样，以下仅列出部分：

- `(let x ([y1 e1] [y2 e2] ...) ...body)` ->
  `(letrec ([x (lambda (y1 y2 ...) ...body)]) (x e1 e2 ...))`
- `(letrec ([x1 e1] [x2 e2] ...) ...body)` ->
  `(let ([x1 (void)] [x2 (void)] ...) (set! x1 e1) (set! x2 e2) ...body)`
- `(let ([x1 e1] [x2 e2] ...) ...body)` ->
  `((lambda (x1 x2 ...) ..body) e1 e2 ...)`

该部分实现于`transform.js`的`desugar`函数中。实现中利用到了[matches.js](matches.js)，这是一个JS的模式匹配库，被本项目用来高效地编写基于JSON的中间表示的变换；该库已用npm安装至本项目中，无需额外安装。

形式转换

去除语法糖后，将JSON格式的中间表示转化为scheme的运行时`Value`。

该部分主要实现于`transform.js`中的`json2scm`函数里。

执行求值

参照前文所述的核心语义，对scheme `Value`进行求值。

该部分主要实现在`evaluate.js`中的`evaluate`函数里。

库函数

在`lib.scm`中用scheme编写了一系列较为基本的功能性函数，如`map`、`for-each`、`filter`、`range`等。