Assignment 1 (Due: Wednesday (12:00pm (צהריים), November 8, 2017)

Mayer Goldberg

October 29, 2017

Contents

1 General

- You may work on this assignment alone, or with a single partner. If you are unable to find or maintain a partner with whom to work on the assignment, then you will work on it alone.
- You are not to copy/reuse code from other students, students of previous years, or code you found on the internet. If you do, you are guilty of *academic dishonesty*. All discovered cases of *academic dishonesty* will be forwarded to *va'adat mishma'at* (ועדת משמעת) for disciplinary action. Please save yourself the trouble, and save us the effort and the inconvenience of dealing with you on this level. This is not how we prefer to relate to our students, but *academic dishonesty* is a real problem, and if necessary, we will pursue all disciplinary venues available to us.
- You should be very careful to test your work before you submit. Testing your work means that all the files you require are available and usable and are included in your submission.
- Make sure your code doesn't generate any unnecessary output: Forgotten debug statements, unnecessary print statements, etc., will result in your output being considered incorrect, and you will lose points. You will not get back these points by appealing or writing emails and complaints, so **please** be careful and make sure your code runs properly.

- Please read this document completely, from start to finish, before beggining work on the assignment.

2 An Extended Reader for Scheme

In this assignment, you will create a Scheme procedure `<sexpr>`, which implements a *reader* for the extended language of S-expressions, and answers the parser interface given within the *parsing combinator* package we posted on the course website.

2.1 The extended Syntax

Your reader will support nearly all of the syntax of S-expressions in Scheme. You will not support the full numerical tower, but rather only integers and fractions. Additionally, you will support two interesting extensions to the syntax of S-expressions:

1. Infix notation for arithmetic operations. Traditionally, the parentheses based prefix notation has proven to be a powerful syntactic notation for code and data. However, the syntax falls short when handling arithmetic and array expressions, proving rather cumbersome. The extended syntax you must support proposes to allow the best of both worlds by using a special *escape syntax* to allow embedding infix expressions within a larger S-expression. The exact same *escape syntax* allows for embedding prefix sub-expressions within larger infix expressions. with this extension, prefix and infix expressions can be intermixed at any depth of nesting.

2. Special notation that shall provide syntactic support for the *call-by-name* mechanism for parameter-passing. The actual implementation of *call-by-name* shall, of course, happen in the code generator, but we add the syntactic support early on.

The grammar you will need to support is roughly[1] the following:[2]

⟨*Sexpr*⟩	::= ⟨*Boolean*⟩ | ⟨*Char*⟩ | ⟨*Number*⟩ | ⟨*String*⟩
	| ⟨*Symbol*⟩ | ⟨*ProperList*⟩ | ⟨*ImproperList*⟩
	| ⟨*Vector*⟩ | ⟨*Quoted*⟩ | ⟨*QuasiQuoted*⟩
	| ⟨*Unquoted*⟩ | ⟨*UnquoteAndSpliced*⟩
	| ⟨*CBName*⟩ | ⟨*InfixExtension*⟩
⟨*Boolean*⟩	::= `#f` | `#t`
⟨*Char*⟩	::= ⟨*CharPrefix*⟩ (⟨*VisibleSimpleChar*⟩
	| ⟨*NamedChar*⟩ | ⟨*HexUnicodeChar*⟩)
⟨*CharPrefix*⟩	::= `#\`
⟨*VisibleSimpleChar*⟩	::= `c`, where `c` is a character that is greater than the space character in the ASCII table
⟨*NamedChar*⟩	::= `lambda`, `newline`, `nul`, `page`, `return`, `space`, `tab`
⟨*HexUnicodeChar*⟩	::= `x` ⟨*HexChar*⟩$^+$
⟨*HexChar*⟩	::= `0`| $\cdots$ | `9`| `a`| $\cdots$ | `f`
⟨*Number*⟩	::= ⟨*Integer*⟩ | ⟨*Fraction*⟩
⟨*Integer*⟩	::= `(+|-)`$^?$ ⟨*Natural*⟩

[1]Caveats and details are noted in subsequent subsections.

[2]If e is an expression, e^* stands for a catenation of zero or more occurrences of e, e^+ stands for a catenation of one or more occurrences of e, and $e^?$ stands for either zero or one occurrences of e.

⟨*Natural*⟩ ::= (`0`| ⋯ |`9`)$^+$
⟨*Fraction*⟩ ::= ⟨*Integer*⟩ `/` ⟨*Natural*⟩
⟨*String*⟩ ::= `"` ⟨*StringChar*⟩* `"`
⟨*StringChar*⟩ ::= ⟨*StringLiteralChar*⟩ | ⟨*StringMetaChar*⟩
    | ⟨*StringHexChar*⟩
⟨*StringLiteralChar*⟩ ::= `c`, where `c` is *any* character other than the backslash character (`\`) or the double-quote char (`"`)
⟨*StringMetaChar*⟩ ::= `\\`| `\"`| `\t`| `\f`| `\n`| `\r`
⟨*StringHexChar*⟩ ::= `\x` ⟨*HexChar*⟩* `;`
⟨*Symbol*⟩ ::= ⟨*SymbolChar*⟩$^+$
⟨*SymbolChar*⟩ ::= (`0` | ⋯ | `9`) | (`a` | ⋯ | `z`) | (`A` | ⋯ | `Z`) | `!` | `$`
    | `^` | `*` | `-` | `_` | `=` | `+` | `<` | `>` | `?` | `/`
⟨*ProperList*⟩ ::= `(` ⟨*Sexpr*⟩* `)`
⟨*ImproperList*⟩ ::= `(` ⟨*Sexpr*⟩$^+$ `.` ⟨*Sexpr*⟩ `)`
⟨*Vector*⟩ ::= `#(` ⟨*Sexpr*⟩* `)`
⟨*Quoted*⟩ ::= `'` ⟨*Sexpr*⟩
⟨*QuasiQuoted*⟩ ::= `` ` `` ⟨*Sexpr*⟩
⟨*Unquoted*⟩ ::= `,` ⟨*Sexpr*⟩
⟨*UnquoteAndSpliced*⟩ ::= `,@` ⟨*Sexpr*⟩

⟨*CBName*⟩ ::= ⟨*CBNameSyntax1*⟩ | ⟨*CBNameSyntax2*⟩
⟨*CBNameSyntax1*⟩ ::= `@` ⟨*Sexpr*⟩
⟨*CBNameSyntax2*⟩ ::= `{` ⟨*Sexpr*⟩ `}`

⟨*InfixExtension*⟩ ::= ⟨*InfixPrefixExtensionPrefix*⟩
    ⟨*InfixExpression*⟩
⟨*InfixPrefixExtensionPrefix*⟩ ::= `##` | `#%`
⟨*InfixExpression*⟩ ::= ⟨*InfixAdd*⟩ | ⟨*InfixNeg*⟩ | ⟨*InfixSub*⟩
    | ⟨*InfixMul*⟩ | ⟨*InfixDiv*⟩ | ⟨*InfixPow*⟩
    | ⟨*InfixArrayGet*⟩ | ⟨*InfixFuncall*⟩
    | ⟨*InfixParen*⟩ | ⟨*InfixSexprEscape*⟩
    | ⟨*InfixSymbol*⟩ | ⟨*Number*⟩
⟨*InfixSymbol*⟩ ::= ⟨*Symbol*⟩ other than `+`, `-`, `*`, `**`, `^`, `/`.
⟨*InfixAdd*⟩ ::= ⟨*InfixExpression*⟩ `+` ⟨*InfixExpression*⟩
⟨*InfixNeg*⟩ ::= `-` ⟨*InfixExpression*⟩
⟨*InfixSub*⟩ ::= ⟨*InfixExpression*⟩ `-` ⟨*InfixExpression*⟩
⟨*InfixMul*⟩ ::= ⟨*InfixExpression*⟩ `*` ⟨*InfixExpression*⟩
⟨*InfixDiv*⟩ ::= ⟨*InfixExpression*⟩ `/` ⟨*InfixExpression*⟩
⟨*InfixPow*⟩ ::= ⟨*InfixExpression*⟩ ⟨*PowerSymbol*⟩
    ⟨*InfixExpression*⟩
⟨*PowerSymbol*⟩ ::= `^` | `**`
⟨*InfixArrayGet*⟩ ::= ⟨*InfixExpression*⟩ `[` ⟨*InfixExpression*⟩ `]`
⟨*InfixFuncall*⟩ ::= ⟨*InfixExpression*⟩ `(` ⟨*InfixArgList*⟩ `)`
⟨*InfixArgList*⟩ ::= ⟨*InfixExpression*⟩ (`,` ⟨*InfixExpression*⟩)* | ε
⟨*InfixParen*⟩ ::= `(` ⟨*InfixExpression*⟩ `)`
⟨*InfixSexprEscape*⟩ ::= ⟨*InfixPrefixExtensionPrefix*⟩ ⟨*Sexpr*⟩

2.1.1 Extensions in support of infix notation

This grammar is *ambiguous* in its infix portion, because we did not specify *precedence*. We expect you to make sure that the expected precedence rules for standard infix programming languages, such as C & Java are maintained. This means that you will need to modify the productions to guarantee that multiplication has higher precedence than addition, that exponentiation has higher precedence than multiplication, and that infix expressions *associate* correctly, so that, for example, `2 - 3 - 4` is converted to `(- (- 2 3) 4)`, etc.

Infix expressions should be converted to equivalent expressions in Scheme. Examples:

- The infix expression `#%2+3-5` should be converted to `(- (+ 2 3) 5)`.
- Function calls of the form `##f(x, y, z)` should be converted to `(f x y z)`.
- Array dereferences should be converted to appropriate `vector-ref` expressions, which can be nested (!). For example, `#%A[1]+A[2]*A[3]^B[4][5][6]` should be converted to: `(+ (vector-ref a 1) (* (vector-ref A 2) (expt (vector-ref A 3) (vector-ref (vector-ref (vector-ref B 4) 5) 6))))`.

There are many possibilities, and you should test each kind against the parser we uploaded. Please remember that infix expressions can contain prefix sub-expressions, and vice versa, arbitrarily-deep. So, for example:

```
> (test-string <sexpr>
  "


(cons
    #;this-is-in-infix
    #%f(x+y, x-z, x*t,
#;this-is-in-prefix
#%(g (cons x y)
     #;#%this-is-in-infix
     #%cons(x, y)
     #;#%this-is-in-infix
     #%list(x, y)
     #;#%this-is-in-infix
     #%h(#;this-is-in-prefix
 #%(* x y),
 #;this-is-in-prefix
 #%(expt x z))))
    #%2)
")
((match (cons
  (f (+ x y)
     (- x z)
     (* x t)
     (g (cons x y)
(cons x y)
(list x y)
(h (* x y)
   (expt x z))))
  2))
  (remaining ""))
```

2.1.2 Extensions in support of *call-by-name*

S-expressions that are syntactically tagged with *call-by-name* syntax, will be converted to (`cbname` ⟨*Sexpr*⟩).

```
> (test-string <sexpr> "@a")
((match (cbname a)) (remaining ""))
> (test-string <sexpr> "@(* a b)")
((match (cbname (* a b))) (remaining ""))
> (test-string <sexpr> "{(* a b)}")
((match (cbname (* a b))) (remaining ""))
> (test-string <sexpr> "{  (* a b)  }")
((match (cbname (* a b))) (remaining ""))
> (test-string <sexpr> "{ #%a + b * c ^ d  }")
((match (cbname (+ a (* b (expt c d))))) (remaining ""))
> (test-string <sexpr> " @  #%a + b * c ^ d  ")
((match (cbname (+ a (* b (expt c d))))) (remaining ""))
```

You are only required to support the syntax for *call-by-name* in *prefix* expressions. If you want to use them in infix expressions, you can switch temporarily, as the following examples demonstrate:

```
> (test-string <sexpr> "#%f(x, #%@#%x + 1)")
((match (f x (cbname (+ x 1)))) (remaining ""))
> (test-string <sexpr> "#%f(i, #%@#%A[i], #%@#%A[i + 1]^i)")
((match (f i
   (cbname (vector-ref a i))
   (cbname (expt (vector-ref a (+ i 1)) i))))
  (remaining ""))
```

2.2 Case sensitivity

Expressions are meant to be *case-insensitive*, that is `#t` and `#T` are meant to be the same, as well as `#\space` and `#\SPACE`, etc. The **only** expressions that are *case-sensitive* are:

- ⟨*VisibleSimpleChar*⟩
- ⟨*StringLiteralChar*⟩

For these, the case should remain as the user has entered them.

2.3 Comments

You will need to support two kinds of comments both for infix and prefix expressions:

- **Line comments:** These start with the `;` symbol, and continue either to the *end of line* or to the *end of file*, whichever comes sooner.
- **Expression comments:** These start with prefix `#;` and continue for the following expression. Needless to say, such comments can be nested, and can be applied interchangeably to both prefix and infix sub-expressions.

2.4 Whitespace

Whitespaces are all intra-token characters that are less than or equal to the *space character*, i.e., that have ASCII value of 32 or less.

2.5 Examples

```
> (test-string <sexpr> "

(let* ((d ##sqrt(b ^ 2 - 4 * a * c))
       (x1 ##((-b + d) / (2 * a)))
       (x2 ##((-b - d) / (2 * a))))
  `((x1 ,x1) (x2 ,x2)))

")
((match (let* ((d (sqrt (- (expt b 2) (* (* 4 a) c))))
       (x1 (/ (+ (- b) d) (* 2 a)))
       (x2 (/ (- (- b) d) (* 2 a))))
  `((x1 ,x1) (x2 ,x2))))
  (remaining ""))

> (test-string <sexpr> "

(let ((result ##a[0] + 2 * a[1] + 3 ^ a[2] - a[3] * b[i][j][i + j]))
  result)

")
((match (let ((result (- (+ (+ (vector-ref a 0)
       (* 2 (vector-ref a 1)))
    (expt 3 (vector-ref a 2)))
 (* (vector-ref a 3)
    (vector-ref
      (vector-ref (vector-ref b i) j)
      (+ i j))))))
  result))
  (remaining ""))

> (test-string <sexpr> "

(let ((result (* n ##3/4^3 + 2/7^5)))
  (* result (f result) ##g(g(g(result, result), result), result)))

")
((match (let ((result (* n (+ (expt 3/4 3) (expt 2/7 5)))))
  (* result
     (f result)
     (g (g (g result result) result) result))))
  (remaining ""))
```

```
> (test-string <sexpr> "

## a[0] + a[a[0]] * a[a[a[0]]] ^ a[a[a[a[0]]]] ^ a[a[a[a[a[0]]]]]

")
((match (+ (vector-ref a 0)
   (* (vector-ref a (vector-ref a 0))
      (expt
(vector-ref a (vector-ref a (vector-ref a 0)))
(expt
  (vector-ref
    a
    (vector-ref a (vector-ref a (vector-ref a 0))))
  (vector-ref
    a
    (vector-ref
      a
      (vector-ref a (vector-ref a (vector-ref a 0))))))))))
  (remaining ""))

> (test-string <sexpr> "

(define pi ##4 * atan(1))

")
((match (define pi (* 4 (atan 1)))) (remaining ""))

;;; Commenting out an entire arithmetic expression
> (test-string <sexpr> "

## 2 + #; 3 - 4 + 5 * 6 ^ 7 8

")
((match (+ 2 8)) (remaining ""))

> (test-string <sexpr> "

`(the answer is: ##2 * 3 + 4 * 5)

")
((match `(the answer is: (+ (* 2 3) (* 4 5))))
  (remaining ""))

> (test-string <sexpr> "(+ 1 ##2 + 3 a b c)")
((match (+ 1 (+ 2 3) a b c)) (remaining ""))
```

2.6 Submission Guidelines

In this course, we use the `git` DVCS for assignment publishing and submission. You can find more information on `git` at `https://git-scm.com/`.

To begin your work, clone the assignment template from the course website:

```
git clone https://www.cs.bgu.ac.il/~comp181/assignments/1
```

This will create a copy of the assignment template folder, named `1`, in your local directory. The template contains three (3) files:

- `sexpr-parser.scm` (the assignment interface)
- `pc.scm` (the *parsing combinators* library)
- `readme.txt`

The file `sexpr-parser.scm` is the interface file for your assignment. The definitions in this file will be used to test your code. If you make breaking changes to these definitions, we will be unable to test and grade your assignment. Do not break the interface. Operations which are considered interface-breaking:

- Modifying the first line: `(load "pc.scm")`
- Modifying the third line: `(define <sexpr>`

Other than breaking the interface, you are allowed to add any code and/or files you like.

Among the files you are required to edit is the file `readme.txt`.

The file `readme.txt` should contain

1. The names and IDs of all the people who worked on this assignment. There should be either your own name, or your name and that of your partner. You may only have one partner for this assignment.
2. The following statement:

   I (We) assert that the work we submitted is 100% our own. We have not received any part from any other student in the class, nor have we give parts of it for use to others. Nor have we used code from other sources: Courses taught previously at this university, courses taught at other universities, various bits of code found on the internet, etc.

   We realize that should our code be found to contain code from other sources, that a formal case shall be opened against us with ועדת משמעת, in pursuit of disciplinary action.

Submissions are only allowed through the submission system.

You are required to submit a **patch file** of the changes you made to the assignment template. See instructions on how to create a patch file below.

Please be careful to check your work multiple times. Because of the size of the class, we cannot handle appeals to recheck your work in case you forget or fail to follow any instructions precisely. Specifically, before you submit your final version, please take the time to make sure your code loads and runs properly in a fresh Scheme session.

2.6.1 Creating a patch file

Before creating the patch review the change list and make sure it contains all the changes that you applied and noting more. Modified files are automatically detected by git but new files must be added explicitly with the 'git add' command:

```
git add -Av .; git commit -m "write a commit message"
```

At this point you may review all the changes you made (the patch):

```
git diff origin
```

Once you are ready to create a patch, simply make sure the output is redirected to the patch file:

```
git diff origin > assigment1.patch
```

After submission (but before the deadline), it is strongly recommended that you download, apply and test your submitted patch file. Assuming you download `assignment1.patch` to your home directory, this can be done in the following manner:

```
cd ~
git clone https://www.cs.bgu.ac.il/~comp181/assignments/1 fresh_assignment1
cd fresh_assignment1
git apply ~/assignment1.patch
```

Then test the result in the directory `fresh_assignment1`.

Finally, remember that your work will be tested on lab computers only! We advise you to test your code on lab computers prior to submission!