Assignment 1 (Due: Wednesday (12:00pm (צהריים), November 23, 2016)

Mayer Goldberg

November 15, 2016

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

2 An Extended Reader for Scheme

Write 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 an interesting extension to the syntax of S-expressions: 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.

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

$$
\begin{aligned}
\langle \mathit{Sexpr} \rangle &::= \langle \mathit{Boolean} \rangle \mid \langle \mathit{Char} \rangle \mid \langle \mathit{Number} \rangle \mid \langle \mathit{String} \rangle \\
&\quad \mid \langle \mathit{Symbol} \rangle \mid \langle \mathit{ProperList} \rangle \mid \langle \mathit{ImproperList} \rangle \\
&\quad \mid \langle \mathit{Vector} \rangle \mid \langle \mathit{Quoted} \rangle \mid \langle \mathit{QuasiQuoted} \rangle \\
&\quad \mid \langle \mathit{Unquoted} \rangle \mid \langle \mathit{UnquoteAndSpliced} \rangle \\
&\quad \mid \langle \mathit{InfixExtension} \rangle \\
\langle \mathit{Boolean} \rangle &::= \texttt{\#f} \mid \texttt{\#t} \\
\langle \mathit{Char} \rangle &::= \langle \mathit{CharPrefix} \rangle \ (\ \langle \mathit{VisibleSimpleChar} \rangle \\
&\quad \mid \langle \mathit{NamedChar} \rangle \mid \langle \mathit{HexUnicodeChar} \rangle \ ) \\
\langle \mathit{CharPrefix} \rangle &::= \texttt{\#\textbackslash} \\
\langle \mathit{VisibleSimpleChar} \rangle &::= \texttt{c}, \text{ where } \texttt{c} \text{ is a character that is greater than} \\
&\quad \text{the space character in the ASCII table} \\
\langle \mathit{NamedChar} \rangle &::= \texttt{lambda}, \texttt{newline}, \texttt{nul}, \texttt{page}, \texttt{return}, \texttt{space}, \\
&\quad \texttt{tab} \\
\langle \mathit{HexUnicodeChar} \rangle &::= \texttt{x} \ \langle \mathit{HexChar} \rangle^{+} \\
\langle \mathit{HexChar} \rangle &::= \texttt{0} \mid \cdots \mid \texttt{9} \mid \texttt{a} \mid \cdots \mid \texttt{f} \\
\langle \mathit{Number} \rangle &::= \langle \mathit{Integer} \rangle \mid \langle \mathit{Fraction} \rangle \\
\langle \mathit{Integer} \rangle &::= (\texttt{+} \mid \texttt{-})^{?} \ \langle \mathit{Natural} \rangle \\
\langle \mathit{Natural} \rangle &::= (\texttt{0} \mid \cdots \mid \texttt{9})^{+} \\
\langle \mathit{Fraction} \rangle &::= \langle \mathit{Integer} \rangle \ \texttt{/} \ \langle \mathit{Natural} \rangle \\
\langle \mathit{String} \rangle &::= \texttt{"} \ \langle \mathit{StringChar} \rangle^{*} \ \texttt{"} \\
\langle \mathit{StringChar} \rangle &::= \langle \mathit{StringLiteralChar} \rangle \mid \langle \mathit{StringMetaChar} \rangle \\
&\quad \mid \langle \mathit{StringHexChar} \rangle \\
\langle \mathit{StringLiteralChar} \rangle &::= \texttt{c}, \text{ where } \texttt{c} \text{ is } \textit{any} \text{ character other than the} \\
&\quad \text{backslash character } (\texttt{\textbackslash}) \\
\langle \mathit{StringMetaChar} \rangle &::= \texttt{\textbackslash\textbackslash} \mid \texttt{\textbackslash"} \mid \texttt{\textbackslash t} \mid \texttt{\textbackslash f} \mid \texttt{\textbackslash n} \mid \texttt{\textbackslash r}
\end{aligned}
$$

[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.

$$
\begin{aligned}
\langle \mathit{StringHexChar}\rangle &::= \texttt{\textbackslash x}\ \langle \mathit{HexChar}\rangle^{*}\ \texttt{;}\\
\langle \mathit{Symbol}\rangle &::= \langle \mathit{SymbolChar}\rangle^{+}\\
\langle \mathit{SymbolChar}\rangle &::= (\texttt{0} \mid \cdots \mid \texttt{9}) \mid (\texttt{a} \mid \cdots \mid \texttt{z}) \mid (\texttt{A} \mid \cdots \mid \texttt{Z}) \mid \texttt{!} \mid \texttt{\$}\\
&\quad \mid \texttt{\^{}} \mid \texttt{*} \mid \texttt{-} \mid \texttt{\_} \mid \texttt{=} \mid \texttt{+} \mid \texttt{<} \mid \texttt{>} \mid \texttt{?} \mid \texttt{/}\\
\langle \mathit{ProperList}\rangle &::= \texttt{(}\ \langle \mathit{Sexpr}\rangle^{*}\ \texttt{)}\\
\langle \mathit{ImproperList}\rangle &::= \texttt{(}\ \langle \mathit{Sexpr}\rangle^{+}\ \texttt{.}\ \langle \mathit{Sexpr}\rangle\ \texttt{)}\\
\langle \mathit{Vector}\rangle &::= \texttt{\#(}\ \langle \mathit{Sexpr}\rangle^{*}\ \texttt{)}\\
\langle \mathit{Quoted}\rangle &::= \texttt{'}\ \langle \mathit{Sexpr}\rangle\\
\langle \mathit{QuasiQuoted}\rangle &::= \texttt{`}\ \langle \mathit{Sexpr}\rangle\\
\langle \mathit{Unquoted}\rangle &::= \texttt{,}\ \langle \mathit{Sexpr}\rangle\\
\langle \mathit{UnquoteAndSpliced}\rangle &::= \texttt{,@}\ \langle \mathit{Sexpr}\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle \mathit{InfixExtension}\rangle &::= \langle \mathit{InfixPrefixExtensionPrefix}\rangle\\
&\quad\ \langle \mathit{InfixExpression}\rangle\\
\langle \mathit{InfixPrefixExtensionPrefix}\rangle &::= \texttt{\#\#} \mid \texttt{\#\%}\\
\langle \mathit{InfixExpression}\rangle &::= \langle \mathit{InfixAdd}\rangle \mid \langle \mathit{InfixNeg}\rangle \mid \langle \mathit{InfixSub}\rangle\\
&\quad \mid \langle \mathit{InfixMul}\rangle \mid \langle \mathit{InfixDiv}\rangle \mid \langle \mathit{InfixPow}\rangle\\
&\quad \mid \langle \mathit{InfixArrayGet}\rangle \mid \langle \mathit{InfixFuncall}\rangle\\
&\quad \mid \langle \mathit{InfixParen}\rangle \mid \langle \mathit{InfixSexprEscape}\rangle\\
&\quad \mid \langle \mathit{InfixSymbol}\rangle \mid \langle \mathit{Number}\rangle\\
\langle \mathit{InfixSymbol}\rangle &::= \langle \mathit{Symbol}\rangle \text{ other than } \texttt{+},\ \texttt{-},\ \texttt{*},\ \texttt{**},\ \texttt{\^{}},\ \texttt{/}.\\
\langle \mathit{InfixAdd}\rangle &::= \langle \mathit{InfixExpression}\rangle\ \texttt{+}\ \langle \mathit{InfixExpression}\rangle\\
\langle \mathit{InfixNeg}\rangle &::= \texttt{-}\ \langle \mathit{InfixExpression}\rangle\\
\langle \mathit{InfixSub}\rangle &::= \langle \mathit{InfixExpression}\rangle\ \texttt{-}\ \langle \mathit{InfixExpression}\rangle\\
\langle \mathit{InfixMul}\rangle &::= \langle \mathit{InfixExpression}\rangle\ \texttt{*}\ \langle \mathit{InfixExpression}\rangle\\
\langle \mathit{InfixDiv}\rangle &::= \langle \mathit{InfixExpression}\rangle\ \texttt{/}\ \langle \mathit{InfixExpression}\rangle\\
\langle \mathit{InfixPow}\rangle &::= \langle \mathit{InfixExpression}\rangle\ \langle \mathit{PowerSymbol}\rangle\\
&\quad\ \langle \mathit{InfixExpression}\rangle\\
\langle \mathit{PowerSymbol}\rangle &::= \texttt{\^{}} \mid \texttt{**}\\
\langle \mathit{InfixArrayGet}\rangle &::= \langle \mathit{InfixExpression}\rangle\ \texttt{[}\ \langle \mathit{InfixExpression}\rangle\ \texttt{]}\\
\langle \mathit{InfixFuncall}\rangle &::= \langle \mathit{InfixExpression}\rangle\ \texttt{(}\ \langle \mathit{InfixArgList}\rangle\ \texttt{)}\\
\langle \mathit{InfixArgList}\rangle &::= \langle \mathit{InfixExpression}\rangle\ (\texttt{,}\ \langle \mathit{InfixExpression}\rangle)^{*} \mid \varepsilon\\
\langle \mathit{InfixParen}\rangle &::= \texttt{(}\ \langle \mathit{InfixExpression}\rangle\ \texttt{)}\\
\langle \mathit{InfixSexprEscape}\rangle &::= \langle \mathit{InfixPrefixExtensionPrefix}\rangle\ \langle \mathit{Sexpr}\rangle
\end{aligned}
$$

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.

Corrections (20161115):

- The state ⟨*StringVisibleChar*⟩ was renamed to ⟨*StringLiteralChar*⟩.
- The definition of ⟨*StringLiteralChar*⟩ now permits *any* character that is not the backslash (`\`). This is a **great** simplification of your work, and also corrects some snags in the definition.
- The Definition of ⟨*HexUnicodeChar*⟩ was changed to require *at least one Hex digit*. This means you change `*star` to `*plus` in your code and everything else works.

Clarification (20161114): 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 ""))
```

Correction (20161111): Changed the rule for ⟨*InfixExpression*⟩ to include productions for ⟨*InfixSymbol*⟩ and ⟨*Number*⟩. Thanks to **eladal** for pointing out the problem!

Correction (20161108): The rule for ⟨*InfixFuncall*⟩ has been changed.

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 What to hand in

For this problem, add your code to the file called `compiler.scm`. Congratulations! You have completed the first stage in the pipeline of your compiler. **Please keep this file for future assignments.**

Correction (20161108): Please create and submit an archive `hw1.zip` that shall unzip to the following directory tree:

- `hw1/`
  - `readme.txt`
  - `compiler.scm`
  - `pc.scm`

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.

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 these instructions precisely. Specifically, before you submit your final version, please take the time to unpack the archive, and make sure your code loads and runs with the above directory structure. For example, make sure your code can access the file `pc.scm`!