Assignment 1 (Due: Sunday at 12:00pm (צהריים), November 21, 2021)

Mayer Goldberg

November 6, 2021

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 an ocaml procedure `nt_sexpr`, that 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 (on Moodle).

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, fractions, and floating-point numbers.

Additionally, you will support two interesting extensions to the syntax of S-expressions:

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

$\langle Sexpr\rangle$ ::=	$\langle Boolean\rangle \mid \langle Char\rangle \mid \langle Number\rangle \mid \langle String\rangle$
	$\mid \langle Symbol\rangle \mid \langle ProperList\rangle \mid \langle ImproperList\rangle$
	$\mid \langle Vector\rangle \mid \langle Quoted\rangle \mid \langle QuasiQuoted\rangle$
	$\mid \langle Unquoted\rangle \mid \langle UnquoteAndSpliced\rangle$
$\langle Boolean\rangle$ ::=	`#f` | `#t`
$\langle Char\rangle$ ::=	$\langle CharPrefix\rangle$ ($\langle VisibleSimpleChar\rangle$
	| $\langle NamedChar\rangle$)
$\langle CharPrefix\rangle$ ::=	`#\`
$\langle VisibleSimpleChar\rangle$ ::=	`c`, where `c` is a visible character, that is a character without a graphical representation, which is an ASCII character that is greater than the *space-character* (ASCII 32)
$\langle NamedChar\rangle$ ::=	`newline`, `nul`, `page`, `return`, `space`, `tab`
$\langle HexChar\rangle$ ::=	`0`| ⋯ | `9`| `a`| ⋯ | `f`
$\langle Number\rangle$ ::=	$\langle Integer\rangle \mid \langle Fraction\rangle \mid \langle Float\rangle$
$\langle Integer\rangle$ ::=	(`+`|`-`)$^?$ $\langle Natural\rangle$
$\langle Natural\rangle$ ::=	(`0`|⋯|`9`)$^+$
$\langle Fraction\rangle$ ::=	$\langle Integer\rangle$ `/` $\langle Natural\rangle$
$\langle Float\rangle$ ::=	(`+`|`-`)$^?$ ($\langle FloatA\rangle \mid \langle FloatB\rangle \mid \langle FloatC\rangle$)
$\langle FloatA\rangle$ ::=	$\langle IntegerPart\rangle$ `.` $\langle Mantissa\rangle^?$ $\langle Exponent\rangle^?$
$\langle FloatB\rangle$ ::=	`.` $\langle Mantissa\rangle$ $\langle Exponent\rangle^?$
$\langle FloatC\rangle$ ::=	$\langle IntegerPart\rangle$ $\langle Exponent\rangle$
$\langle IntegerPart\rangle$ ::=	(`0`|⋯|`9`)$^+$
$\langle Mantissa\rangle$ ::=	(`0`|⋯|`9`)$^+$
$\langle Exponent\rangle$ ::=	$\langle ExponentToken\rangle$ $\langle Integer\rangle$
$\langle ExponentToken\rangle$ ::=	`e` | `E` | `*10^` | `*10**`
$\langle String\rangle$ ::=	`"` $\langle StringChar\rangle^*$ `"`
$\langle StringChar\rangle$ ::=	$\langle StringLiteralChar\rangle \mid \langle StringMetaChar\rangle$
	$\mid \langle StringHexChar\rangle \mid \langle StringInterpolated\rangle$
$\langle StringLiteralChar\rangle$ ::=	`c`, where `c` is *any* character other than the backslash character (`\`) or the double-quote char (`"`) or the tilde (`~`)

[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.

⟨*StringMetaChar*⟩::=`\\`| `\"`| `\t`| `\f`| `\n`| `\r` | `~~`
⟨*StringHexChar*⟩::=`\x` ⟨*HexChar*⟩$^+$ `;`
⟨*StringInterpolated*⟩::= `~{`⟨*sexpr*⟩`}` See details below.
⟨*Symbol*⟩::=⟨*SymbolChar*⟩$^+$
⟨*SymbolChar*⟩::=(`0` | ··· | `9`) | (`a` | ··· | `z`) | (`A` | ··· | `Z`) | `!` | `$`
| `^` | `*` | `-` | `_` | `=` | `+` | `<` | `>` | `?` | `/` | `:`
⟨*ProperList*⟩::=`(` ⟨*Sexpr*⟩* `)`
⟨*ImproperList*⟩::=`(` ⟨*Sexpr*⟩$^+$ `.` ⟨*Sexpr*⟩ `)`
⟨*Vector*⟩::=`#(` ⟨*Sexpr*⟩* `)`
⟨*Quoted*⟩::=`'` ⟨*Sexpr*⟩
⟨*QuasiQuoted*⟩::=`` ` `` ⟨*Sexpr*⟩
⟨*Unquoted*⟩::=`,` ⟨*Sexpr*⟩
⟨*UnquoteAndSpliced*⟩::=`,@` ⟨*Sexpr*⟩

The abstract syntax of sexprs is defined via the following ocaml types:

```
type scm_number =
  | ScmRational of (int * int)
  | ScmReal of float;;

type sexpr =
  | ScmVoid
  | ScmNil
  | ScmBoolean of bool
  | ScmChar of char
  | ScmString of string
  | ScmSymbol of string
  | ScmNumber of scm_number
  | ScmVector of (sexpr list)
  | ScmPair of (sexpr * sexpr);;
```

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 three 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.
- **Matching comments:** In principle, text between matching curly braces `{...}` is commented out. Matching comments may nest, so curly braces must match. However curly braces that are part of tokens, such as strings that contain curly braces, or the curly-brace characters (`#\{`, `#\}`) are not counted when balancing matching comments.

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 Interpolated Strings

Some programming languages, such as Python and Javascript, extend the syntax of strings to a form of *templates*, permitting one to intermix static text with dynamically-computed text. This is what is known as *interpolated strings*, and is very similar to how *quasiquote-expressions* are used to intermix static and dynamic data within lists and vectors.

For this assignment, you shall extend the syntax of strings to permit another, special *meta-character*: The *tilde* (`~`). Of course, to use the tilde-character itself within a string, you should double it, so that `~~` should expand to the `#\~` character within the string.

The syntax for embedding dynamic text is given as `~{`⟨*sexpr*⟩`}`. The ⟨*sexpr*⟩, which may be surrounded by arbitrary whitespace and comments, is assumed to be the concrete syntax for Scheme code. The code is evaluated, and converted into a string using the `format` procedure. For example: `~{ (+ 2 3) }"` should be replaced with the *sexpr* `(format "~a" (+ 2 3))`, which returns a string. This string is then appended (at run-time!) to the static parts of the surrounding string using the procedure `string-append`. Please consult the examples below to see the data-structures returned.

2.6 Examples

```
# test_string nt_sexpr "#t" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 2; found = ScmBoolean true}
# test_string nt_sexpr "#T" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 2; found = ScmBoolean true}
# test_string nt_sexpr "#f" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 2; found = ScmBoolean false}
# test_string nt_sexpr "#F" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 2; found = ScmBoolean false}
# test_string nt_sexpr "#\\a" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 3; found = ScmChar 'a'}
# test_string nt_sexpr "#\\A" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 3; found = ScmChar 'A'}
# test_string nt_sexpr "\"\"" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 2; found = ScmString ""}
# test_string nt_sexpr "\"moshe!\"" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 8; found = ScmString "moshe!"}
# test_string nt_sexpr "\"moshe!\\n\\t\\r\\f\"" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 16; found = ScmString "moshe!\n\t\r\012"}
# test_string nt_sexpr "\"The letter 'a' can be entered as \\x61;\""
   0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 40;
 found = ScmString "The letter 'a' can be entered as a"}
# test_string nt_sexpr "\"The letter 'A' can be entered as \\x41;\""
   0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 40;
 found = ScmString "The letter 'A' can be entered as A"}
# test_string nt_sexpr "lambda" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 6; found = ScmSymbol "lambda"}
# test_string nt_sexpr "if" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 2; found = ScmSymbol "if"}
# test_string nt_sexpr "#\\space" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 7; found = ScmChar ' '}
# test_string nt_sexpr "#\\return" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 8; found = ScmChar '\r'}
# test_string nt_sexpr "#\\newline" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 9; found = ScmChar '\n'}
# test_string nt_sexpr "#\\tab" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 5; found = ScmChar '\t'}
# test_string nt_sexpr "#\\ponpon" 0;;
Exception: PC.X_no_match.
# test_string nt_sexpr "#\\gafrur" 0;;
Exception: PC.X_no_match.
# test_string nt_sexpr "#\\\\" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 3; found = ScmChar '\\'}
# test_string nt_sexpr "#\\\"" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 3; found = ScmChar '"'}
```

```
# test_string nt_sexpr "#\\x41" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 5; found = ScmChar 'A'}
# test_string nt_sexpr "#\\x20" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 5; found = ScmChar ' '}
# test_string nt_sexpr "#\\x61" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 5; found = ScmChar 'a'}
# test_string nt_sexpr "1234" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 4; found = ScmNumber (ScmRational (1234, 1)
   )}
# test_string nt_sexpr "00001234" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 8; found = ScmNumber (ScmRational (1234, 1)
   )}
# test_string nt_sexpr "00001234e0" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 10; found = ScmNumber (ScmReal 1234.)}
# test_string nt_sexpr "2/3" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 3; found = ScmNumber (ScmRational (2, 3))}
# test_string nt_sexpr "2/0" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 3; found = ScmSymbol "2/0"}
# test_string nt_sexpr "2/6" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 3; found = ScmNumber (ScmRational (1, 3))}
# test_string nt_sexpr "1.234" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 5; found = ScmNumber (ScmReal 1.234)}
# test_string nt_sexpr "1.234e1" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 7; found = ScmNumber (ScmReal 12.34)}
# test_string nt_sexpr "1.234e+1" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 8; found = ScmNumber (ScmReal 12.34)}
# test_string nt_sexpr "1.234*10^+1" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 11; found = ScmNumber (ScmReal 12.34)}
# test_string nt_sexpr "1.234*10^1" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 10; found = ScmNumber (ScmReal 12.34)}
# test_string nt_sexpr "1.234*10^-1" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 11;
 found = ScmNumber (ScmReal 0.12340000000000001)}
```

```
# test_string nt_sexpr ".1234e-10" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 9; found = ScmNumber (ScmReal 1.234e-11)}
# test_string nt_sexpr ".1234*10**-10" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 13; found = ScmNumber (ScmReal 1.234e-11)}
# test_string nt_sexpr ".1234*10^-10" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 12; found = ScmNumber (ScmReal 1.234e-11)}
# test_string nt_sexpr "-.1234*10^-10" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 13; found = ScmNumber (ScmReal (-1.234e-11)
   )}
# test_string nt_sexpr "()" 0;;
- : sexpr PC.parsing_result = {index_from = 0; index_to = 2; found =
   ScmNil}
# test_string nt_sexpr "#()" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 3; found = ScmVector []}
# test_string nt_sexpr "(1 . 2)" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 7;
 found =
  ScmPair (ScmNumber (ScmRational (1, 1)), ScmNumber (ScmRational (2,
     1)))}
# test_string nt_sexpr "(1.2)" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 5;
 found = ScmPair (ScmNumber (ScmReal 1.2), ScmNil)}
# test_string nt_sexpr "#(1.2)" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 6; found = ScmVector [ScmNumber (ScmReal
   1.2)]}
# test_string nt_sexpr "#(1 2)" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 6;
 found =
  ScmVector [ScmNumber (ScmRational (1, 1)); ScmNumber (ScmRational (2,
      1))]}
# test_string nt_sexpr "#(a b c)" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 8;
 found = ScmVector [ScmSymbol "a"; ScmSymbol "b"; ScmSymbol "c"]}
# test_string nt_sexpr "(a b c)" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 7;
 found =
  ScmPair
```

```
   (ScmSymbol "a", ScmPair (ScmSymbol "b", ScmPair (ScmSymbol "c",
      ScmNil)))}
# test_string nt_sexpr "(a b . c)" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 9;
 found = ScmPair (ScmSymbol "a", ScmPair (ScmSymbol "b", ScmSymbol "c")
    )}
# test_string nt_sexpr "((a . #t) (b . #f))" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 19;
 found =
  ScmPair
   (ScmPair (ScmSymbol "a", ScmBoolean true),
    ScmPair (ScmPair (ScmSymbol "b", ScmBoolean false), ScmNil))}
# test_string nt_sexpr "   #(   )   " 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 12; found = ScmVector []}
# test_string nt_sexpr "    (   )   " 0;;
- : sexpr PC.parsing_result = {index_from = 0; index_to = 12; found =
   ScmNil}
# test_string nt_sexpr "(define a 3)" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 12;
 found =
  ScmPair
   (ScmSymbol "define",
    ScmPair (ScmSymbol "a", ScmPair (ScmNumber (ScmRational (3, 1)),
       ScmNil)))}
# test_string nt_sexpr "\"~{(+ 2 3)}\"" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 12;
 found =
  ScmPair
   (ScmSymbol "format",
    ScmPair
     (ScmString "~a",
      ScmPair
       (ScmPair
         (ScmSymbol "+",
          ScmPair
           (ScmNumber (ScmRational (2, 1)),
            ScmPair (ScmNumber (ScmRational (3, 1)), ScmNil))),
        ScmNil)))}
# test_string nt_sexpr "\"~{   (+ 2 3)    }\"" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 19;
 found =
  ScmPair
```

```
   (ScmSymbol "format",
    ScmPair
     (ScmString "~a",
      ScmPair
       (ScmPair
         (ScmSymbol "+",
          ScmPair
           (ScmNumber (ScmRational (2, 1)),
            ScmPair (ScmNumber (ScmRational (3, 1)), ScmNil))),
         ScmNil)))}
# test_string nt_sexpr "`(,a ,@b)" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 9;
 found =
  ScmPair
   (ScmSymbol "quasiquote",
    ScmPair
     (ScmPair
       (ScmPair (ScmSymbol "unquote", ScmPair (ScmSymbol "a", ScmNil)),
        ScmPair
         (ScmPair
           (ScmSymbol "unquote-splicing", ScmPair (ScmSymbol "b",
              ScmNil)),
          ScmNil)),
      ScmNil))}
# test_string nt_sexpr "'a" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 2;
 found = ScmPair (ScmSymbol "quote", ScmPair (ScmSymbol "a", ScmNil))}
# test_string nt_sexpr "''a" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 3;
 found =
  ScmPair
   (ScmSymbol "quote",
    ScmPair
     (ScmPair (ScmSymbol "quote", ScmPair (ScmSymbol "a", ScmNil)),
        ScmNil))}
# test_string nt_sexpr "'''a" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 4;
 found =
  ScmPair
   (ScmSymbol "quote",
    ScmPair
     (ScmPair
       (ScmSymbol "quote",
        ScmPair
```

```
          (ScmPair (ScmSymbol "quote", ScmPair (ScmSymbol "a", ScmNil)),
           ScmNil)),
       ScmNil))}
# test_string nt_sexpr "```a" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 4;
 found =
  ScmPair
   (ScmSymbol "quasiquote",
    ScmPair
     (ScmPair
       (ScmSymbol "quasiquote",
        ScmPair
         (ScmPair (ScmSymbol "quasiquote", ScmPair (ScmSymbol "a",
            ScmNil)),
          ScmNil)),
      ScmNil))}
# test_string nt_sexpr ",@a" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 3;
 found =
  ScmPair (ScmSymbol "unquote-splicing", ScmPair (ScmSymbol "a", ScmNil
     ))}
# test_string nt_sexpr ",@,@,@a" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 7;
 found =
  ScmPair
   (ScmSymbol "unquote-splicing",
    ScmPair
     (ScmPair
       (ScmSymbol "unquote-splicing",
        ScmPair
         (ScmPair
           (ScmSymbol "unquote-splicing", ScmPair (ScmSymbol "a",
              ScmNil)),
          ScmNil)),
      ScmNil))}
# test_string nt_sexpr "((lambda (x) `(,x ',x)) '(lambda (x) `(,x ',x))
   )" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 48;
 found =
  ScmPair
   (ScmPair
     (ScmSymbol "lambda",
      ScmPair
       (ScmPair (ScmSymbol "x", ScmNil),
```

```
    ScmPair
     (ScmPair
       (ScmSymbol "quasiquote",
        ScmPair
         (ScmPair
           (ScmPair
             (ScmSymbol "unquote", ScmPair (ScmSymbol "x", ScmNil))
                ,
            ScmPair
             (ScmPair
               (ScmSymbol "quote",
                ScmPair
                 (ScmPair
                   (ScmSymbol "unquote", ScmPair (ScmSymbol "x",
                      ScmNil)),
                  ScmNil)),
              ScmNil)),
          ScmNil)),
      ScmNil))),
ScmPair
 (ScmPair
   (ScmSymbol "quote",
    ScmPair
     (ScmPair
       (ScmSymbol "lambda",
        ScmPair
         (ScmPair (ScmSymbol "x", ScmNil),
          ScmPair
           (ScmPair
             (ScmSymbol "quasiquote",
              ScmPair
               (ScmPair
                 (ScmPair
                   (ScmSymbol "unquote", ScmPair (ScmSymbol "x",
                      ScmNil)),
                  ScmPair
                   (ScmPair
                     (ScmSymbol "quote",
                      ScmPair
                       (ScmPair
                         (ScmSymbol "unquote",
                          ScmPair (ScmSymbol "x", ScmNil)),
                        ScmNil)),
                    ScmNil)),
                ScmNil)),
            ScmNil))),
      ScmNil)),
  ScmNil))}
```

```
# test_string nt_sexpr "\"2 + 3 = ~{(+ 2 3)}\"" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 20;
 found =
  ScmPair
   (ScmSymbol "string-append",
    ScmPair
     (ScmString "2 + 3 = ",
      ScmPair
       (ScmPair
         (ScmSymbol "format",
          ScmPair
           (ScmString "~a",
            ScmPair
             (ScmPair
               (ScmSymbol "+",
                ScmPair
                 (ScmNumber (ScmRational (2, 1)),
                  ScmPair (ScmNumber (ScmRational (3, 1)), ScmNil))),
               ScmNil))),
        ScmNil)))}
# test_string nt_sexpr "\"This is static: ABC and this is dynamic: ~{\"
   even though the string is static *in Scheme*, it is interpolated, so
    we consider it dynamic...\"}\"" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 142;
 found =
  ScmPair
   (ScmSymbol "string-append",
    ScmPair
     (ScmString "This is static: ABC and this is dynamic: ",
      ScmPair
       (ScmPair
         (ScmSymbol "format",
          ScmPair
           (ScmString "~a",
            ScmPair
             (ScmString
               "even though the string is static *in Scheme*, it is
                  interpolated, so we consider it dynamic...",
               ScmNil))),
        ScmNil)))}
# test_string nt_sexpr "\"static ~{'dynamic} more static ~{'(more
   dynamic!)} \"" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 53;
 found =
  ScmPair
```

```
   (ScmSymbol "string-append",
    ScmPair
     (ScmString "static ",
      ScmPair
       (ScmPair
         (ScmSymbol "format",
          ScmPair
           (ScmString "~a",
            ScmPair
             (ScmPair
               (ScmSymbol "quote", ScmPair (ScmSymbol "dynamic", ScmNil
                  )),
              ScmNil))),
        ScmPair
         (ScmString " more static ",
          ScmPair
           (ScmPair
             (ScmSymbol "format",
              ScmPair
               (ScmString "~a",
                ScmPair
                 (ScmPair
                   (ScmSymbol "quote",
                    ScmPair
                     (ScmPair
                       (ScmSymbol "more",
                        ScmPair (ScmSymbol "dynamic!", ScmNil)),
                      ScmNil)),
                  ScmNil))),
            ScmPair (ScmString " ", ScmNil))))))}
# test_string nt_sexpr "

;;; This is a line comment!
#;\"and this is an S-expression (string) that is removed via a sexpr-
   comment!\"

(a b c
 mary had a little lambda!
 #;#;#;#;\"I bet you didn't realize that sexpr-comments\"
 \"may be\" \"nested!\"
 \"so all four strings shall be dumped and not appear in the list!\"
)

" 0;;
                        - : sexpr PC.parsing_result =
{index_from = 0; index_to = 290;
 found =
  ScmPair
```

```
   (ScmSymbol "a",
    ScmPair
     (ScmSymbol "b",
      ScmPair
       (ScmSymbol "c",
        ScmPair
         (ScmSymbol "mary",
          ScmPair
           (ScmSymbol "had",
            ScmPair
             (ScmSymbol "a",
              ScmPair
               (ScmSymbol "little", ScmPair (ScmSymbol "lambda!",
                  ScmNil))))))))}
# test_string nt_sexpr "(you should know {that you can have paired/
   matching comments too, and that these are entered using braces!})"
   0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 108;
 found =
  ScmPair
   (ScmSymbol "you",
    ScmPair (ScmSymbol "should", ScmPair (ScmSymbol "know", ScmNil)))}
# test_string nt_sexpr "({and {that {these too}}} may be nested!)" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 41;
 found =
  ScmPair
   (ScmSymbol "may",
    ScmPair (ScmSymbol "be", ScmPair (ScmSymbol "nested!", ScmNil)))}
# test_string nt_sexpr "(a {#\\}} b c)" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 13;
 found =
  ScmPair
   (ScmSymbol "a", ScmPair (ScmSymbol "b", ScmPair (ScmSymbol "c",
      ScmNil)))}
# test_string nt_sexpr "(a {#\\{} b c)" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 13;
 found =
  ScmPair
   (ScmSymbol "a", ScmPair (ScmSymbol "b", ScmPair (ScmSymbol "c",
      ScmNil)))}
# test_string nt_sexpr "(a {\"}}}}{{{{\"} b c)" 0;;
- : sexpr PC.parsing_result =
{index_from = 0; index_to = 20;
 found =
  ScmPair
   (ScmSymbol "a", ScmPair (ScmSymbol "b", ScmPair (ScmSymbol "c",
      ScmNil)))}
```

3 Final Words

You are given two files for this assignment:

- `pc.ml`, which is the *parsing combinators* package
- `reader.ml`, which is a skeleton file containing the definitions and some code to help you along the way. Under no circumstances should you change the type definitions in this file. You may add additional types if you like, but ultimately, you must work with the `scm_number` and `sexpr` data types that are given to you. If you fail to do so, your code shall surely fail the automated tests!

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.