Assignment 1 (Due: Tuesday, 13/11/18, Noon 12:00PM)

Mayer Goldberg

November 11, 2018

Contents

1 General

- You may work on this assignment alone, or with a single partner. You may not join a group of two or more students to work on the assignment. 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* 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
- Your code should generate absolutely no warnings or error messages. If it does, you get a grade of zero.
- No late submissions will be accepted, no exceptions. A late submission is, **by definition**, *any time* past the official deadline, according to the departmental *submission system*.
- Please read this document completely, from start to finish, before beginning work on the assignment.

2 Introduction

For this assignment, you shall write a reader for Scheme. This is not a difficult assignment, but it is a lot of work, and you should start early and test frequently.

There are two parts to this assignment: (1) constructing a parser given a language specification (in the form of a CFG); and (2) extending a parser given additional syntactic requirements, but without a formal specification. The purpose of the second half is to allow you to think about and experiment with defining your own grammars. You are required to complete both parts.

Your final parser will not support all of the concrete syntax supported by Chez Scheme, and there are some non-standard extensions you will implement that are not supported by Chez Scheme (these syntactic forms will be pointed out in the assignment description).

You are given a template file `reader.ml` for this assignment. If you miss any parts of this assignment, you will need to complete them in time for the final project submission, since this assignment is an important part of your complete compiler.

Please remember *not to change the signatures and types of the modules in `reader.ml` *. Changing these signatures will make the automated tests fail and "award" you a grade of zero. No appeals will be accepted for assignments with modified signatures or types.

In order to help you make sure that you did not make any breaking changes, we will provide you with a "structure test". Always run the structure test before submission.

3 A *reader* (parser for S-expressions (sexprs))

In this part of the assignment you will write the first part of the sexpr parser, which should accept the language defined by the CFG provided below. Along with the next part of the assignment, the result will constitute the first part of the pipeline of your Scheme compiler, the *reader*.

The reader is a parser for sexprs: It reads text from a string, and outputs an *Abstract Syntax Tree* for sexprs. You will need to use the *parsing combinators* package (provided on the course website) to write mutually-referential parsers for the various kinds of sexprs, and use these to define two procedures:

- `read_sexpr : string -> sexpr`
- `read_sexprs : string -> sexpr list`

The first procedure takes a string and assumes it contains one (and only one) sexpr, whereas the second procedure takes a string and assumes it contains any number of sexprs. Both functions must return ASTs which correctly represent the input. Such ASTs are represented by the `sexpr` type defined in `reader.ml`.

As you recognize different kinds of sexprs, you will need to provide a *pack* wrapper for the parsers, that will invoke the type constructors for the corresponding `sexpr` sub-types.

As seen in class, the `sexpr` type is a disjoint type of atomic and composite sub-types:

```
type sexpr =
  | Bool of bool
  | Nil
  | Number of number
  | Char of char
  | String of string
  | Symbol of string
  | Pair of sexpr * sexpr
  | Vector of sexpr list;;
```

Please note that we shall not support the full numeric tower of Scheme. Instead, we support only two types: integers and floating-point numbers. The type of `number` encapsulates both sub-types, and is given in `reader.ml`:

```
type number =
  | Int of int
  | Float of float;;
```

3.1 The CFG

The following CFG describes the first part of the language your parser should support. Following the CFG definition is an extended discussion of the various forms and sub-types of sexpr which appear in this CFG.

$$\begin{aligned}\langle \mathit{Sexpr}\rangle ::= {}& \langle \mathit{Boolean}\rangle \mid \langle \mathit{Char}\rangle \mid \langle \mathit{Number}\rangle \mid \langle \mathit{String}\rangle \\ & \mid \langle \mathit{Symbol}\rangle \mid \langle \mathit{List}\rangle \mid \langle \mathit{DottedList}\rangle \mid \langle \mathit{Vector}\rangle \\ & \mid \langle \mathit{Quoted}\rangle \mid \langle \mathit{QuasiQuoted}\rangle \mid \langle \mathit{Unquoted}\rangle \\ & \mid \langle \mathit{UnquoteAndSpliced}\rangle\end{aligned}$$

⟨*Boolean*⟩	::= `#f` | `#t`
⟨*Char*⟩	::= ⟨*CharPrefix*⟩ (⟨*VisibleSimpleChar*⟩ | ⟨*NamedChar*⟩ | ⟨*HexChar*⟩)
⟨*CharPrefix*⟩	::= `#\`
⟨*HexPrefix*⟩	::= `#x`
⟨*VisibleSimpleChar*⟩	::= `c`, where `c` is a character that is greater than the space character in the ASCII table
⟨*NamedChar*⟩	::= `newline`, `nul`, `page`, `return`, `space`, `tab`
⟨*HexChar*⟩	::= `x` ⟨*HexDigit*⟩$^+$
⟨*HexDigit*⟩	::= (`0`| ⋯| `9`) | (`a`| ⋯| `f`) | (`A`| ⋯| `F`)
⟨*Digit*⟩	::= (`0`| ⋯| `9`)
⟨*Number*⟩	::= ⟨*Integer*⟩ | ⟨*Float*⟩ | ⟨*HexInteger*⟩ | ⟨*HexFloat*⟩
⟨*Integer*⟩	::= (`+`|`-`)$^?$ ⟨*Natural*⟩
⟨*HexInteger*⟩	::= ⟨*HexPrefix*⟩ (`+` | `-`)$^?$ ⟨*HexNatural*⟩
⟨*Natural*⟩	::= ⟨*Digit*⟩$^+$
⟨*HexNatural*⟩	::= ⟨*HexDigit*⟩$^+$
⟨*Float*⟩	::= ⟨*Integer*⟩ `.` ⟨*Natural*⟩
⟨*HexFloat*⟩	::= ⟨*HexInteger*⟩ `.` ⟨*HexNatural*⟩
⟨*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` ⟨*HexDigit*⟩$^+$ `;`
⟨*Symbol*⟩	::= ⟨*SymbolChar*⟩$^+$
⟨*SymbolChar*⟩	::= (`0` | ⋯ | `9`) | (`a` | ⋯ | `z`) | (`A` | ⋯ | `Z`) | `!` | `$` | `^` | `*` | `-` | `_` | `=` | `+` | `<` | `>` | `?` | `/` | `:`
⟨*List*⟩	::= `(` ⟨*Sexpr*⟩* `)`
⟨*DottedList*⟩	::= `(` ⟨*Sexpr*⟩$^+$ `.` ⟨*Sexpr*⟩ `)`
⟨*Vector*⟩	::= `#(` ⟨*Sexpr*⟩* `)`
⟨*Quoted*⟩	::= `'` ⟨*Sexpr*⟩
⟨*QuasiQuoted*⟩	::= `` ` `` ⟨*Sexpr*⟩
⟨*Unquoted*⟩	::= `,` ⟨*Sexpr*⟩
⟨*UnquoteAndSpliced*⟩	::= `,@` ⟨*Sexpr*⟩

3.2 Comments & whitespaces

An sexpr may contain whitespace characters and comments. Your parser will have to know to skip over these as it constructs the AST for the sexpr it's reading.

3.2.1 Whitespaces

Any character with an ASCII value less than or equal to the *space* character, is considered a whitespace for the purpose of this assignment. Whitespaces may appear wherever Chez Scheme accepts them.

3.2.2 Line comments

Line comments start with the semicolon character `;` and continue until either an *end-of-line* or *end-of-input* is reached. The semicolon may appear anywhere on the line, and need not be the first character!

This kind of comment is used to document your code.

3.2.3 Sexpr comments

Scheme includes another kind of comment, not so much to document your code as to "hide" the next sexpr without actually removing it from the source file. This kind of "commenting out" is very handy when debugging code.

Sexpr comments start with the characters `#;` and continue to the end of the following sexpr. There may be no space or comment between the `#` and the `;` characters. This sexpr can be any valid sexpr: A number, a Boolean, a symbol, but more often than not, a deeply-nested expression with balanced parentheses.

Please note that while removing comments is usually "easy", sexpr-comments are non-trivial, because they rely on the ability of your parser to recognize what is a valid sexpr! This is your first production that is *recursive*.

Note that sexpr comments may be used in succession to 'remove' consecutive sexprs. For example, the expression: `#;#;(+ 1 2) 1 "Only one left"` evaluates to the string `"Only one left"`.

3.3 The concrete syntax of sexprs

3.3.1 Boolean

There are two Boolean values in Scheme: `#f` (for *false*), and `#t` (for *true*). Your reader should recognize these in a case-insensitive manner, so that `#t` and `#T` are treated the same, etc.

3.3.2 Number

Scheme has a rich numerical tower. We will not be supporting the full numerical tower in our compiler, but we do want to experience the polymorphism of Scheme procedures, so we will support two kinds of numbers: integers & floating-point numbers.

1. Integers

   Integers can be positive or negative. They may begin with any number of zeros. They may come with an optional sign, which can either be positive or negative. Here are examples of valid integers in our language:

   - `1234`
   - `01234`
   - `-1234`
   - `-012`
   - `-0`
   - `+1234`
   - `+432`

We shall also allow numbers to be entered in hexadecimal (base 16), with the prefix `#x`. A couple of things to note:

- In the case of hexadecimal numbers, the sign comes **after** the `#x` prefix.
- Both the prefix `#x` and the values `a-f` are case insensitive, meaning that any mixture of capital and non-capital letters is allowed.

Here are some examples of valid integers in hexadecimal:

- `#x1234`
- `#Xabcd`
- `#x-234bf`
- `#X+AbCd12`

2. Floating-point numbers Floating point numbers are inexact representations of real values. Like integers, floating-point numbers may begin with any number of zeros and may include an optional sign (positive or negative). Additionally, they may include any number of trailing zeroes after the decimal point. Here are some examples of valid floating-point numbers in our language:

   - `1.0`
   - `0005.0129`
   - `501.100000000000000000000`
   - `-0.0`
   - `+999.12349999999`
   - `-102.000000000000001`

   In practice, there is a limit on the number of digits following the decimal point. In the interest of simplicity, your parser will allow an unlimited number of digits, and rely on OCaml to truncate the extra digits.

3.3.3 Symbol

In an interactive Scheme system, a symbol is represented internally as a *hashed string*. Our compiler, however, is not an *interactive* (or *online*) compiler, but a *batch* (or *offline*) one, so the AST for symbols will just contain the string used to represent the symbol.

In principle, a symbol in Scheme can contain any character. This is the case for symbols that have been created *dynamically* from strings, using `string->symbol`. Constant symbols that are read in by the reader obey a stricter syntax. The characters in a symbol may include the following characters:

- The lowercase letters: `a`, … , `z`
- The uppercase letters: `A`, … , `Z`
- Digits: `0`, … , `9`
- Punctuation: `!$^*-_=+<>/?`

Scheme has traditionally been *case-insensitive* with respect to symbols, although this has changed in R^{6}RS (the 6th version of the Scheme standard). For the purpose of this course, you must support case-insensitivity for symbols in the following way: Your parser should convert all literal symbol characters to lowercase. Hence the symbols `abc`, `Abc`, `aBc`, and `ABC` are all the same object, which is stored internally as `abc`.

3.3.4 String

Strings are monomorphic arrays of characters. Syntactically, strings in Scheme are delimited by double-quote marks, may contain any character, and span across several lines. Here are some examples of strings:

```
"moshe"

"a string"

"This is a very long
string that spills across
several lines."
```

1. String meta-chars

   Some characters cannot appear in a string without being preceded by a special *backslash* prefix. This is the exact same situation as in C, C++, Java, Python, and many other programming languages. The list of meta-characters you need to support is:

Char	ASCII	Concrete syntax
return	ASCII 13	`\r`
newline	ASCII 10	`\n`
tab	ASCII 9	`\t`
page	ASCII 12	`\f`
backslash	ASCII 92	`\\`
double quote	ASCII 34	`\"`

2. Hexadecimal string chars Strings may contain characters represented as hexadecimal ASCII codes. In this case the token is prefixed by the two characters `\x` and is terminated by a `;`. The hexadecimal char representation covers both meta and literal string chars. For example:

   - `\x30;` is the literal string char `0`
   - `\xa;` is the meta char `\n`

   Since your parser (and eventually, your compiler) is only required to support ASCII chars, you may assume that we will only test your parser with ASCII hexadecimal values.

3.3.5 Char

Characters are denoted by the character prefix `#\`. There can be no whitespace or comments between the the `#` & `\` characters. What follows the prefix falls under one of several categories:

1. Named chars

   Some characters are denoted by their full name:

Char	ASCII	Concrete syntax
nul	ASCII 0	`#\nul`
newline	ASCII 10	`#\newline`
return	ASCII 13	`#\return`
tab	ASCII 9	`#\tab`
formfeed	ASCII 12	`#\page`
space	ASCII 32	`#\space`

   Named characters are case insensitive, so that `#\page`, `#\Page`, and `#\PAGE` are all denote the same character.

2. Visible chars

   Characters that are in the visible range (i.e., have an ASCII value that is larger than 32) can be entered "as-is", with the character prefix. Here are some examples:

   - `#\a`
   - `#\A`
   - `#\?`

   Notice that visible characters are case-sensitive, so that `#\a` & `#\A` are distinct.

3. Hexadecimal chars A third way to input chars in sexprs is as hexadecimal values. In this case the prefix consists of three characters: `#\x`. The hexadecimal char representation covers both named and visible chars. For example:

   - `#\x30` is the visible char `#\0`
   - `#\xa` is the named char `#\newline`

   Since your parser (and eventually, your compiler) is only required to support ASCII chars, you may assume that we will only test your parser with ASCII hexadecimal values.

3.3.6 Nil

Nil, or the empty list, is simply a matching pair of parentheses: `()`. These may enclose whitespaces and comments.

3.3.7 Pair

The concrete syntax for pairs respects the two *dot rules*, generating *proper lists* and *improper lists*. The formal grammar for these is as follows:

$$\begin{array}{rcl} \langle\text{List}\rangle & ::= & (\ \langle\text{Sexpr}\rangle^{*}\ ) \\ \langle\text{DottedList}\rangle & ::= & (\ \langle\text{Sexpr}\rangle^{+}\ .\ \langle\text{Sexpr}\rangle\ ) \end{array}$$

You will need to convert both proper and improper lists to nested instances of the `Pair` sub-type.

3.3.8 Vector

The grammar for the concrete syntax of vectors is similar to the first production for pairs:

$$\langle\text{Vector}\rangle \quad ::= \quad \#(\ \langle\text{Sexpr}\rangle^* \ )$$

Notice that vectors too are recursive.

3.3.9 Quote-like forms

You need to support 4 quote-like forms:

$$\begin{aligned}
\langle\text{Quoted}\rangle &::= \ \texttt{'}\langle\text{Sexpr}\rangle \\
\langle\text{QQuoted}\rangle &::= \ \text{`}\,\langle\text{Sexpr}\rangle \\
\langle\text{UnquotedSpliced}\rangle &::= \ ,@\,\langle\text{Sexpr}\rangle \\
\langle\text{Unquoted}\rangle &::= \ ,\langle\text{Sexpr}\rangle
\end{aligned}$$

For these, you should generate the sexprs `Pair(Symbol(name), Pair(sexpr, Nil()))`, where `name` is one of `quote`, `quasiquote`, `unquote-splicing`, and `unquote`, respectively, and `sexpr` is the sexpr that follows the quote-like tag.

4 Extending your parser without a formal specification

4.1 Scientific notation

Your parser must also support numbers (both integer and floating-point) given in scientific notation. Valid numbers given in scientific notation begin with either a valid integer or a valid floating point number, followed by either the character `e` or the character `E`, followed by a valid integer. Here are some examples of valid numbers in scientific notation in our language:

- `1e1`
- `1E+1`
- `10e-1`
- `3.14e+9`
- `3.14E-512`
- `+000000012.3E00000002`
- `-5.000000000e-2`

Note that both integers and floating-point numbers may be given in scientific notation. You are expected to output the correct numeric sub-type for a valid number in scientific notation in the final AST.

4.2 Bracket notation for pairs

Some of the sexpr-based languages (for example, Scheme) support the use of square brackets (`[` and `]`) to denote pairs (proper- and improper-lists), in addition to parentheses. Your parser must also support this syntax. Note that an opening square bracket must be matched with a closing square bracket, and an opening parentheses must be matched with a closing parentheses. This means that the following examples are syntactically incorrect:

```
(+ 1 2]

(let ([x 1)
      (y 2])
  [+ x y]]
```

4.3 Automatic balancing of parentheses

As an extension to the concrete syntax of sexprs, your parser should support a special "close all open parentheses" operator: `...` (three dots without separating spaces or comments).

This syntactic extension should work for any valid mixture of parentheses `(` and brackets `[`.

Note that the `...` syntax may be used in conjunction with partially-balanced expressions. For example:

```
(let ([x 1]
      [y 2])
  [lambda (z)
    (+ x y z) ...
```

In this snippet, `...` should only balance the opening parenthesis of the `lambda` and the opening parenthesis of the whole `let` expression.

4.4 A final note regarding case sensitivity

Most 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, be it upper-case or lower-case.

5 What to submit

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/~comp191/compiler`

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

- `reader.ml` (the assignment interface)

- `pc.ml` (the *parsing combinators* library)
- `readme.txt`

The file **`reader.ml`** 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 line: `#use "pc.ml"`
- Modifying the module signatures and types defined in the file

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 *va'adat mishma'at*, 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 note that any modifications you make to `pc.ml` will be discarded during the testing process, as our testers will use our version of `pc.ml` for the tests.

You are provided with a structure test in order to help you ensure that our tests are able to run on your code properly. Make sure to run this test on your final submission.

5.1 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 > compiler.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/~comp191/compiler fresh_compiler
cd fresh_compiler
git apply ~/compiler.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!

6 Tests, testing, correctness, completeness

Please start working on the assignment ASAP! Please keep in mind that:

- We will not (and cannot) provide you with anything close to full coverage. It's your responsibility to test and debug and fix your own code! We're nice, we're helpful, and we'll do our best to provide you with many useful tests, but this is not and cannot be all that you rely on!
- We encourage you to contribute and share tests! Please do not share ocaml code.
- This assignment can be tested using ocaml and Chez Scheme. You don't really need more, beyond a large batch of tests...
- We encourage you to compile your code and test frequently.

7 A word of advice

The class is very large. We do not have the human resources to handle late submissions or late corrections from people who do not follow instructions. By contrast, it should take you very little effort to make sure your submission conforms to what we ask. If you fail to follow the instructions to the letter, you will not have another chance to submit the assignment: If files your work depends on are missing, if functions don't work as they are supposed to, if the statement asserting authenticity of your work is missing, if your work generates output that is not called for (e.g., because of leftover debug statements), etc., then you're going to have points deducted. The graders are instructed not to accept any late corrections or re-submissions under any circumstances.

7.1 A final checklist

1. You completed the module skeleton provided in the `reader.ml` file
2. You did not change the module signatures
3. Your parser runs correctly under OCaml on the departmental Linux image

4. You completed the readme.txt file that contains the following information: (a) Your name and ID (b) The name and ID of your partner for this assignment, assuming you worked with a partner. (c) A statement asserting that the code you are submitting is your own work, that you did not use code found on the internet or given to you by someone other than the teaching staff or your partner for the assignment.

5. You submitted you work in its entirety