Assignment 2 (Due: Sunday, (12:00pm (צהריים), December 3rd, 2017)

Mayer Goldberg

November 20, 2017

Contents

1 General

2 A tag-parser for Scheme
   2.1 The Core Forms
      2.1.1 Constants
      2.1.2 Variables
      2.1.3 Conditionals
      2.1.4 Disjunctions
      2.1.5 Lambda forms
      2.1.6 Define
      2.1.7 Assignments
      2.1.8 Applications
      2.1.9 Sequences
   2.2 Macro-Expanding Special Forms in Scheme
   2.3 Expanding let, let*, letrec
   2.4 Handling quasiquoted expressions
   2.5 Summary

3 Submission Guidelines
   3.1 Creating a patch file

4 A word of advice
   4.1 A final checklist

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 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.
• Your work should run in Chez Scheme on the departmental Linux image. We will not test your work on other platforms. **Test, test, and test again:** Make sure your work runs correctly under Chez Scheme. We will not allow for re-submissions or corrections after the fact, so please be responsible and test!

• 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 beginning work on the assignment.

2 A tag-parser for Scheme

For this assignment, you need to implement a tag-parser in Scheme. You will implement the procedure `parse` which will take an sexpr as an argument, and return a parsed sexpr. Parsing in this sense means annotating the sexpr with tags, so that the type of expression can be known easily and the various sub-expressions can be gotten at with confidence (that is, without having to check each time to make sure that they syntactically correct and legal), and of course, parsing the sub-expressions as well.

Your tag parser should include the macro-expansion code according to the specifications below.

2.1 The Core Forms

The parser will recognize the following Scheme syntax:

2.1.1 Constants

You need to package constants within `const`-records: If an expression $M$ is a constant, then the parsed expression is `(const $M$). There is one special case here that has to do with quoted expressions. For more information, see below.

Constant expressions can be formed from the following types, according to the examples below:

- Nil: (const ())
- Vectors: (const #(1 2 3 4) 2 3))
- Booleans: (const #f)
- Characters: (const #\a)
- Numbers: (const 34)
- Strings: (const "abc")
- Quoted sexprs: Remember that the quote needs to be removed. Consider the following examples:
> (parse '(quote a))
  (const a)
> (parse '(quote (a b c)))
  (const (a b c))
> (parse '(quote (quote a b c)))
  (const (quote a b c))

Note that this last one is *not* an error; Think why!

### 2.1.2 Variables

You will need to package variables within variable-records: If an expression consists of the non-reserved symbol `M`, then the parsed expression is `(var M)`. Consider the following examples:

> (parse 'abc)
  (var abc)
> (parse '123x)
  (var 123x)

This last example demonstrates the extension you were asked to support of the Scheme syntax: Remember that *officially* a variable name isn’t supposed to begin with a digit, but we aren’t going by this.

Here is the list of symbols you are required to treat as reserved words:

```
(define *reserved-words*
  '(and begin cond define do else if lambda
       let let* letrec or quasiquote unquote
       unquote-splicing quote set!))
```

### 2.1.3 Conditionals

Expressions of the form `(if <test> <dit> <dif>)` are parsed as follows: `(if3 (parse <test>) , (parse <dit>) , (parse <dif>))`:

> (parse '(if a b c))
  (if3 (var a) (var b) (var c))
> (parse '(if (if a b c)
      'x
      '(x y z)))
  (if3 (if3 (var a)
     (var b)
     (var c))
     (const x)
     (const (x y z)))

Expressions of the form `(if <test> <dit>)` are parsed as follows: `(if3 (parse <test>) , (parse <dit>) , (parse (void)))`:

> (parse '(if a b))
The `void` object was not introduced in the lecture & assignment on S-expressions because it really isn’t an S-expression: There is no way to type a constant literal void. However, the `void` object is a constant, and should be parsed accordingly, i.e., packaged in the `const` structure. The `void` object was introduced into Scheme in order to make expressions seem more like statements: Recall that expressions have a value whereas statements do not. Many expressions that have side effects do not return values that are used later on. The `void` object provides a way for such expressions to return a value that is not printed, and therefore seem as if nothing has been returned. The values of expressions of the form `(if <test> <dit>)` are obviously not used for any further computation, and such expressions are most often used in places where the `<dit>` expression contains a side effect, which is why the `void` object serves as an implicit else-clause.

### 2.1.4 Disjunctions

An expression of the form `(or expr1 expr2 expr3 ... )` will be parsed using an or-record as follows: `(or (peExpr1 peExpr2 peExpr3 ... ))`, where `peExpr1`, `peExpr2`, `peExpr3`, etc, are the the parsed expressions corresponding to `expr1`, `expr2`, `expr3`, etc. Consider the following examples:

> (parse '(or (zero? x) (zero? y) (zero? z)))
(or ((applic (var zero?) ((var x)))
    (applic (var zero?) ((var y)))
    (applic (var zero?) ((var z)))))

> (parse '(or (or (f1 x) (f2 y))
    (or (f3 z) (f4 w) (f5 r))
    (and (f6 u) (f7 t))))
(or ((or ((applic (var f1) ((var x)))
    (applic (var f2) ((var y)))))
    (or ((applic (var f3) ((var z)))
    (applic (var f4) ((var w)))))
    (applic (var f5) ((var r))))
    (if3 (applic (var f6) ((var u)))
    (applic (var f7) ((var t)))
    (const #f))))

### 2.1.5 Lambda forms

Three types of lambda-expressions must be supported by the parser:

1. Regular lambda
The most common lambda-expression is of the form

\[(\lambda (v_1 \ldots v_n) e_1 \ldots )\]

where \((v_1 \ldots v_n)\) is a list of zero or more variables, and \(e_1, \ldots\) denotes one or more expressions. The regular lambda expression \((\lambda (x_1 \ldots x_n) e_1 \ldots e_m)\) is parsed using a \texttt{lambda-simple} record, as \((\texttt{lambda-simple} (x_1 \ldots x_n) \texttt{<Body>})\) where \texttt{<Body>} is the parsed expression corresponding to the sequence \(e_1 \ldots e_m\). Consider the following examples:

\[
> \text{(parse '}(\lambda (x y z)
   (if x y z))\text{)}
\]
\[
(\texttt{lambda-simple} (x y z)
   (if3 (var x)
     (var y)
     (var z)))
\]

\[
> \text{(parse '}(\lambda () a))
\]
\[
(\texttt{lambda-simple} ()
   (var a))
\]

2. Lambda with optional arguments

The lambda-expression with optional arguments is of the form

\[(\lambda (v_1 \ldots v_n . v\text{-rest}) e_1 \ldots e_m)\]

where \((v_1 \ldots v_n . v\text{-rest})\) is an improper list of variables, and \(e_1 \ldots e_m\) denotes one or more expressions. The intended use of the variable \(v\text{-rest}\) is that upon application it will be bound to the list of optional arguments to the procedure. The lambda expression with optional arguments \((\lambda (x_1 \ldots x_n . x) e_1 \ldots e_m)\) is parsed using a \texttt{lambda-opt} record, as \`(\texttt{lambda-opt} (x_1 \ldots x_n) x ,\text{parse `(begin e_1 \ldots e_m)})\`). Consider the following examples:

\[
> \text{(parse '}(\lambda (x y z . rest)
   (if x y z))\text{)}
\]
\[
(\texttt{lambda-opt} (x y z) \text{rest}
   (if3 (var x)
     (var y)
     (var z)))
\]

\[
> \text{(parse '}(\lambda (x . rest)
   \text{rest})\text{)}
\]
\[
(\texttt{lambda-opt} (x) \text{rest}
   (var \text{rest}))
\]

3. Variadic lambda

The variadic lambda is of the form \((\lambda \text{args} e_1 \ldots e_m)\) where \texttt{args} is a variable, and \(e_1 \ldots e_m\) denotes one or more expressions. The variadic lambda binds the list of its arguments to the single identifier \texttt{args}. The variadic lambda expression \((\lambda \text{args} e_1 \ldots e_m)\) is parsed as a special case of the \texttt{lambda-opt} form, where the list of required parameters is empty: \`(\texttt{lambda-opt} () \text{args} ,\text{parse `(begin e_1 \ldots e_m)})\`). Consider the
You must use the begin special form in order to expand all forms of lambda that have a sequence of more than one expression in the body. This is very different from what you did in PPL, where thunks were used to macro-expand begin-expressions. See the item on sequences below.

2.1.6 Define

Two types of define-expressions should be supported by the parser: “Regular” define-expressions and the MIT-style define-expressions used for defining procedures. They will both be parsed using the single define-record, so the MIT-style define will need some processing so that it fits the correct format.

1. Regular define

An expression of the form (define var expr) will be parsed using the define-record as follows: (define (var var) peExpr) where peExpr is the parsed expression corresponding to expr. Consider the following examples:

> (parse '(define x 5))
(define (var x) (const 5))
> (parse '(define x (lambda (x) x)))
(define (var x)
  (lambda-simple (x)
    (var x)))
2.1.7 Assignments

Assignments are written using set!-expressions. The general form of set!-expressions is (set! v <expr>). This should be parsed into (set (var v) <pExpr>), where <pExpr> is the result of applying the tag-parser to <expr>. For example:

> (parse '(set! x 3))
(set (var x) (const 3))
> (parse '(set! v (f x)))
(set (var v) (applic (var f) ((var x))))

2.1.8 Applications

An expression of the form (E1 ... Em) is an application if E1 isn’t a reserved symbol, and will be parsed using an applic-record as follows: `(applic ,(parse E1) (,(parse E2) ... ,(parse Em))). If E1 is called with no arguments then the list of parsed arguments given by (, (parse E2) ... , (parse Em)) is empty. The rationale behind the structure of the applic-record is that we would like to separate the procedure from the arguments it takes. This will make life a bit simpler later on. Consider the following examples:

> (parse '(a))
(applic (var a) ())
> (parse '(a b c))
(applic (var a)
  ((var b) (var c)))
> (parse '((a b) (a c)))
(applic
  (applic (var a)
    ((var b))
  )
)
2.1.9 Sequences

There are two kinds of sequences of expressions: Explicit, and implicit. Explicit sequences are begin-expressions. Implicit sequences of expressions appear in various special forms, such as cond, lambda, let, etc. Both kinds of sequences will be parsed using the seq record: The sequence $E_1, \ldots, E_n$ of expressions will be parsed as

```scheme
`(seq ,(parse E1) \ldots ,(parse En))
```

2.2 Macro-Expanding Special Forms in Scheme

For this problem you need to make sure you’re accurate and complete. For this problem, you will need to recognize some of Scheme’s special forms and remove them by macro expansion.

The forms you will need to be able to handle are the following:

- `let, let*, letrec`
- `and`
- `cond, MIT-style define expressions`

Please keep in mind that there are many things to check, and that because this is a relatively mindless exercise, your code will be tested for accuracy and thoroughness: The variables on the left hand side of the ribs in a `let` and `letrec` expressions must all be different. This needn’t be the case in `let*` expressions. Read over the Revised Report on Scheme and learn the behavior of the `and` and `or` special forms; It’s not as trivial as it may appear at first. There are many such small points that will need to be checked carefully to make sure that what you have is a valid expression.

Your macro-expanders may not use Scheme’s `gensym` procedure (if you’re not familiar with the `gensym` procedure in Scheme – good for you!) or any other mechanism for inventing new names at run-time (even if you write one yourself!).

2.3 Expanding `let, let*, letrec`

- The order of the ribs in the `let`-expressions and `letrec`-expressions determines the order of the parameters in the corresponding `lambda`-expressions.
- The `let*`-expressions should be expanded into nested `let`-expressions, respecting the order.
- The `letrec`-expressions should be expanded to use side effects in the way described in class.

2.4 Handling `quasiquoted` expressions

Your tag parser will need to be able to handle `quasiquoted` expressions. Expressions of the form `(quasiquote <sexpr>)` will need to be expanded into either constants or applications, based on what they contain. When `<sexpr>` contains sub-expressions of the form `(unquote <sexpr>)`
or \texttt{(unquote-splicing <sexpr>}), you will need to be able to convert such expressions into semantically-equivalent applications using \texttt{cons} and \texttt{append} respectively.

The latest version of Scheme supports \textit{nested quasiquotes}. You do not need to support these, as their implementation is non-trivial. One-level quasiquotation has been present in LISP and Scheme for decades and is quite straightforward to implement. Here are examples from my own \textit{quasiquote expander} (the expander takes the quasiquoted expression without the \texttt{quasiquote} itself):

\begin{verbatim}
> (expand-qq '(a b c)) '(a b c)
> (expand-qq '(a ,b c)) (cons 'a (cons b '(c)))
> (expand-qq '(a ,b ,@c)) (cons 'a (cons b c))
> (expand-qq ',@(a ,b c)) (append a (cons b 'c))
\end{verbatim}

My code includes a very naïve expander for quasiquoted, and a post-expansion optimizer. There’s still plenty of room for further optimization on the expressions I generate, but already they generate reasonable code. Post-expansion optimization’s seem like a far easier route to generating good code for \texttt{quasiquote}-expressions than to try and figure out how to generate perfect code recursively. Some problems just don’t lend themselves to simple recursive solutions.

Please use my code in your parser.

\section*{2.5 Summary}

There is plenty of Scheme syntax that isn’t covered by the parser or the macro system described here. Within this course, we will assume that this extra syntax simply doesn’t exist, and in any case you need not support it in your parser: The parser does not need to recognize any additional syntax – neither additional special forms nor variations on supported forms. Just what’s been specified above.

Your parser should then be able to return parse trees for all the core forms. For the forms that are handled by your macro expander, your parser should expand and then parse these forms, returning parse trees containing the core forms only. This means, for example, that when you parse a \texttt{let}-expression, the parser should return the parse tree for an application of the appropriate parsed lambda and the parsed arguments.

\section*{3 Submission Guidelines}

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

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

\texttt{git clone https://www.cs.bgu.ac.il/~comp181/assignments/2}

This will create a copy of the assignment template folder, named 2, in your local directory. The template contains four (4) files:
• tag-parser.scm (the assignment interface)
• qq.scm (the quasiquote expansion and optimization)
• pattern-matcher.scm (a file required by qq.scm)
• readme.txt

The file **tag-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 "qq.scm")`
- Modifying the third line: `(define parse`

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 ,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.

3.1 Creating a patch file

Before creating the patch review the change set 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:

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

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

```shell
git diff origin
```
Once you are ready to create a patch for submission, simply make sure the output is redirected to the patch file:

```
git diff origin > assignment2.patch
```

After submission (but before the deadline), it is strongly recommended that you download, apply and test your submitted patch file. Assuming you download `assignment2.patch` to your home directory, this can be done in the following manner:

```
cd ~
git clone https://www.cs.bgu.ac.il/~comp181/assignments/2 fresh_assignment2
cd fresh_assignment2
git apply ~/assignment2.patch
```

Then test the result in the directory `fresh_assignment2`.

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!

4 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 you fail to submit a patch file, 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 get a grade of zero. The graders are instructed not to accept any late corrections or re-submissions under any circumstances.

4.1 A final checklist

1. You completed the `parse` function in the `tag-parser.scm` file
2. You did not break the interface
3. Your `tag-parser` supports all the core forms
4. Your `tag-parser` supports all the special forms
5. Your `tag-parser` supports quasiquoted expressions
6. Your `tag-parser` runs correctly under Chez Scheme on the departmental Linux image
7. 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.

8. You committed your work in its entirety

9. You created and submitted a patch file containing your work in the submission system