Assignment 2 (Due: Tuesday, 27/11/18, 10:00AM)

Mayer Goldberg

November 13, 2018

Contents

1 General **1**

2 Introduction **1**

3 A tag-parser (parser for expressions) **2**
3.1 Description 2
3.2 The concrete syntax of Scheme code 2
3.2.1 Core forms 2
3.2.2 Macro-expansions 4

4 What to submit **4**
4.1 Creating a patch file 5
4.2 What we will test 6

5 Tests, testing, correctness, completeness **6**

6 A word of advice **6**
6.1 A final checklist 7

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 tag-parser for Scheme. This is not a difficult assignment, but it is a lot of work, and you should start early and test frequently.

You are given a template file `tag-parser.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 `tag-parser.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 tag-parser (parser for expressions)

3.1 Description

The tag-parser converts from the AST of sexprs to the AST of expressions, and performs macro-expansion along the way. The key procedures you will need to write are `parse_expression : sexpr -> expr` and `parse_expressions : sexpr list -> expr list`.

3.2 The concrete syntax of Scheme code

Your tag-parsering and macro expansnion procedures must behave in the same manner presented in the lectures.

3.2.1 Core forms

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

```
type expr =
  | Const of constant
  | Var of string
  | If of expr * expr * expr
  | Seq of expr list
  | Set of expr * expr
  | Def of expr * expr
  | Or of expr list
  | LambdaSimple of string list * expr
  | LambdaOpt of string list * string * expr
  | Applic of expr * (expr list)

type constant =
  | Sexpr of sexpr
  | Void
```

This type includes only the core forms, other supported forms will be macro-expanded into the corresponding core forms sub-types.

1. Constants: Constants come in two forms: *quoted* and *unquoted*. The field of any *quoted* form is a constant. *Self-evaluating forms* (Booleans, chars, numbers, strings) are constants too, even if they haven't been quoted.

2. Variables The concrete syntax of variables is given as unquoted symbols that are not reserved words. For each variable, you should generate an instance of `Variable`. You are given a list of reserved words at the top of the `Tag_Parser` structure:

   ```
   let reserved_word_list =
     ["and"; "begin"; "cond"; "define"; "else";
      "if"; "lambda"; "let"; "let*"; "letrec"; "or";
      "quasiquote"; "quote"; "set!"; "unquote";
      "unquote-splicing"];;
   ```

   None of these symbols can serve as a variable name (and by extension, as a procedure name).

3. Conditionals: You should support both the *if-then* & *if-then-else* forms of conditions in Scheme. *If-then* forms expand into *if-then-else* forms, where the *else* field has the value `Const (Void)`.

4. Lambda Expressions: There are 3 kinds of λ-expressions in Scheme: *simple*, *with optional arguments*, and *variadic*. We will be using *two* records to represent these three different λ-expressions: `LambdaSimple of string list * expr` and `LambdaOpt of string list * string * expr`. Variadic λ-expressions are represented as `LambdaOpt` structures with an empty list of pramaters.

5. Applications: You should use the type constructor `Applic` to parse applications.

6. Disjunctions: Disjunctions are simply `or`-expressions. Recall that we shall be supporting `or`-expressions as a core form, while macro-expanding `and`-expressions.

7. Definitions: There are two ways to write definitions in Scheme: The basic way, and what I call "the MIT-syntax for define", which is used to define procedures, and which appears throughout the book *The Structure and Interpretation of Computer Programs*. Simple `define` expressions are considered core forms while MIT `define` expressions will be treated as macros.

   Simple `define`-expressions are of the form `(define <name> <expr>)`. The `<name>` is the variable name, and the `<expr>` is the expression the value of which is to be assigned to the given variable. Both the variable name and the expression need to be parsed, giving two values that are instances of `expr` (the first of which must always be a `Variable`). These two values are packaged as an instances of `Def.` arguments.

8. Assignments: Assignments are witten using `set!`-expressions. You should use the type constructor `Set` to parse assignments.

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` type constructor.

   Similar to the handling of `or` and `and` forms, tag-parsed sequences have two base forms which differ from the general tag-parsed form:

   - An empty sequence should be tag-parsed to `Const Void`:
     Evaluating `# Tag_Parser.tag_parse_expression Pair(Symbol "begin", Nil)` should return `expr = Const Void`.
   - A sequence with a single element should not be tag-parsed as a sequence:
     Evaluating `# Tag_Parser.tag_parse_expression Pair(Symbol "begin", Pair(Symbol "a", Nil))` should return `expr = Var "a"`.

   While explicit sequences may be empty, in our course we require that implicit sequences must contain at least one element.

3.2.2 Macro-expansions

1. Quasiquoted expressions: Upon recognizing a *quasiquoted*-expression, you should expand the expression while considering `unquote` and `unquote-splicing` subexpressions. After performing the initial expansion, you need to call the tag-parser recursively over the expanded form.

   Note that you are not required to support nested quasiquote-expressions.

2. `cond`:

   You should expand `cond`-expressions, while supporting the three type of ribs presented in class. The tag-parser should be called recursively over the expanded form.

   Please keep in mind that the `else` keyword is not necessary. A `cond` without an `else` returns `Const (Void)` if none of the tests match (this required should be automatically covered by the expansion of *if-then* to *if-then-else*)

3. `let`:

   Expand these into applications, and call the tag-parsed recursively.

4. `let*`:

   Expand these into nested let expressions, and call the tag-parsed recursively.

5. `letrec`:

   Expand these into `let`-expressions with assignmentes, and call the tag-parsed recursively.

6. `and`:

   Expand these into nested `if`-expressions, and call the tag-parsed recursively.

7. MIT `define`:

   Recall that MIT-style `define`-expressions are of the form `(define(<name> . <argl>) <expr>)`, where

   - `<name>` is the name of the variable
   - `<argl>` represents the list of parameters
   - `<expr>` is an expression.

   The MIT-style `define`-expression should be macro-expanded into an simple `define`-expression containing the relevant lambda form. The tag-parser should be called recursively over the expanded form.

4 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, pull the updated assignment template from the course website by navigating to your local repository folder from assignment 1 and executing the command: `git pull`

If you haven't work on assignment 1 before starting this work, you can simply clone a fresh copy of the repository by executing the command:

```
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 four (4) files:

- `reader.ml` (assignment 1 interface)
- `tag-parser.ml` (assignment 2 interface)
- `pc.ml` (the *parsing combinators* library)
- `readme.txt`

The file `tag-parser.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 "reader.ml"`
- Modifying the types defined in `"reader.ml"`
- Modifying the module signatures and types defined in `"tag-parser.ml"`

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.

4.1 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 `assignment2.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_compiler`.

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!

4.2 What we will test

The patch you submit for this assignment includes your work from assignment 1 and any corrections you made, in addition to your work on assignment 2. Despite this, only your work on this assignment (assignment 2) will be tested and graded. That means we will **not** run your reader to generate sexprs for the tag-parser.

5 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.

6 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.

6.1 A final checklist

1. You completed the module skeleton provided in the `tag-parser.ml` file
2. Your tag-parsering and macro-expanding procedures match the behaviour presented in class
3. You did not change the module signatures
4. Your parser runs correctly under OCaml on the departmental Linux image
5. 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.
6. You submitted you work in its entirety