Assignment 2 (Due: Monday at 12:00pm (צהריים), December 19, 2022)

Mayer Goldberg

December 1, 2022

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 the disciplinary committee (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 beginning work on the assignment.

2 Introduction

This assignment combines the second and third homework assignment into a single assignment with a single, extended deadline. This was done to save time. Please start working on the assignment as early as you can!

You are given the file `hw2-template.ml`. This file is a skeleton file that contains definitions for types, modules, and procedures you need to complete the assignment. You may add your own types and procedures, but please do not modify the signature types you given, as this shall break the grading code. Please combine this file with the code for your reader, and rename the combined file `compiler.ml`.

3 A tag-parser for Scheme

You shall need to complete the code for the tag-parser, most of which has already been given to you. The tag-parser takes the output from your readers (`sexprs`, the *abstract syntax* of data and the *concrete syntax* of expressions), and returns the corresponding *abstract syntax* for the expression (`expr`). The tag-parser has been explained in the lectures and in the slides for the course. Please consult these!

3.1 The `expr` data-type and related types

The type for `expr`, and its related types, are given below, with the exception of `sexpr`, which has been given in previous assignment.

```
type var =
  | Var of string;;

type lambda_kind =
  | Simple
  | Opt of string;;

type expr =
  | ScmConst of sexpr
  | ScmVarGet of var
  | ScmIf of expr * expr * expr
  | ScmSeq of expr list
  | ScmOr of expr list
  | ScmVarSet of var * expr
  | ScmVarDef of var * expr
  | ScmLambda of string list * lambda_kind * expr
  | ScmApplic of expr * expr list;;
```

The abstraction of the type `var` serves several purposes:

- It allows us to restrict the type of the first field in `ScmVarGet`, `ScmVarSet`, and `ScmVarDef` to a variable, rather than have it be `expr`. This gives us extra security because the type-checker will alert us if we accidentally pass a different expression.
- It gives us a nice place, during the semantic analysis phase, to keep additional information on variables, such as the *kind* of variable (*parameter*, *bound*, *free*) and the lexical address.

In some places in the code you are given, the code raises the exception `X_not_yet_implemented`. Your job is to replace such expressions with the complete functionality to implement the tag-parser. When you are done, your tag-parser will be able to parse a large subset of Scheme, including one-level quasiquote-expressions, and several special forms that are supported via macro-expansion.

3.2 How to test your tag-parser

The code you are given includes several tools to help you debug your tag-parser:

- The `sexpr_of_expr` procedure: You are given an "inverse" of sorts to the tag-parser, that takes a parsed expression and returns its concrete-syntax as an `sexpr`.
- The `string_of_expr` procedure: Composing `sexpr_of_expr` with `string_of_sexpr`, we defined the `string_of_expr` procedure. You can use it to get a string representation for your expression.
- The printer-function `print_expr`: This function is used in conjunction with `Printf.printf` to print any user-defined type. To print an expression, you need to use the `%a` formatting string, as in:

  ```
  Printf.printf "The expression is:\n\n%a\n\n" print_expr <some-expression>;;
  ```

3.3 Examples of the tag-parser

```
utop[6]> Tag_Parser.tag_parse (Reader.nt_sexpr "(a b c)" 0).found;;
- : expr =
ScmApplic (ScmVarGet (Var "a"), [ScmVarGet (Var "b"); ScmVarGet (Var "c")])
utop[7]> Tag_Parser.tag_parse (Reader.nt_sexpr
"(if (zero? n) 1 (* n (fact (- n 1))))" 0).found;;
- : expr =
ScmIf (ScmApplic (ScmVarGet (Var "zero?"), [ScmVarGet (Var "n")]),
 ScmConst (ScmNumber (ScmRational (1, 1))),
 ScmApplic (ScmVarGet (Var "*"),
  [ScmVarGet (Var "n");
   ScmApplic (ScmVarGet (Var "fact"),
    [ScmApplic (ScmVarGet (Var "-"),
      [ScmVarGet (Var "n"); ScmConst (ScmNumber (ScmRational (1, 1)))])])]))

utop[9]> let show str = Printf.printf "\n\n%a\n\n"
Tag_Parser.print_expr (Tag_Parser.tag_parse
(Reader.nt_sexpr str 0).found);;
val show : string -> unit = <fun>
```

```
utop[10]> show "(define (fact n) (if (zero? n) 1 (* n (fact (- n 1)))))";;

(define fact (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))
- : unit = ()
utop[11]> show "(let ((a 0) (b 1) (c 2)) (list a b c))";;

(let ((a 0) (b 1) (c 2)) (list a b c))
- : unit = ()
utop[12]> show "(cond ((f? x) 'f) ((g? y) 'g) ((h? z) 'h) (else 'unknown))";;

(if (f? x) 'f (if (g? y) 'g (if (h? z) 'h 'unknown)))
- : unit = ()
utop[13]> show "(lambda (a b . c) (lambda x (lambda (z w) (list a b c x z w))))";;

(lambda (a b . c) (lambda x (lambda (z w) (list a b c x z w))))
- : unit = ()
utop[17]> show "(begin (begin (begin (display \"hello!\") (newline))))";;

(begin (display "hello!") (newline))
- : unit = ()
utop[18]> show "(let* ((a 1) (b 1)
(a (+ a b)) (b (+ a b)) (a (+ a b))
(b (+ a b))) (list a b))";;

(let ((a 1)) (let ((b 1)) (let ((a (+ a b)))
(let ((b (+ a b))) (let ((a (+ a b)))
(let ((b (+ a b))) (list a b)))))))
- : unit = ()
utop[19]> show "(letrec ((fact1 (lambda
```

```
(n) (if (zero? n) 1 (* n (fact2 (- n 1))))))
(fact2 (lambda (n) (if (zero? n) 1 (* n
(fact3 (- n 1))))))) (fact3 (lambda (n)
(if (zero? n) 1 (* n (fact1 (- n 1)))))))
(fact 100))";;


(let ((fact1 'whatever) (fact2 'whatever)
(fact3 'whatever)) (begin (set! fact1
(lambda (n) (if (zero? n) 1 (* n (fact2
(- n 1)))))) (set! fact2 (lambda (n) (if
(zero? n) 1 (* n (fact3 (- n 1))))))
(set! fact3 (lambda (n) (if (zero? n) 1
(* n (fact1 (- n 1)))))) (fact 100)))

- : unit = ()

utop[34]> show "`(a b c)";;


(cons 'a (cons 'b (cons 'c '())))

- : unit = ()

utop[35]> show "`(,a b ,c)";;


(cons a (cons 'b (cons c '())))

- : unit = ()

utop[36]> show "`(,@a b ,@c)";;


(append a (cons 'b c))

- : unit = ()

utop[37]> show "`((a ,a) (b ,b))";;


(cons (cons 'a (cons a '())) (cons (cons 'b (cons b '())) '()))

- : unit = ()

utop[38]> show "\"the value is ~{ (foo x (+ x y))  }\n\"";;
```

```
(string-append "the value is " (format "~a" (foo x (+ x y))) "\n")

- : unit = ()
```

4 The semantic-analysis module

The semantic-analysis module takes a parsed expression and performs several actions on it, some of which add information. To accommodate this extra information, we introduce a type related to `expr`, which we call `expr'`, so that the semantic analysis phase maps from `expr -> expr'`.

4.1 The `expr'` type and related types

Please study the `expr'` type below. The changes between `expr` and `expr'` are intended to accommodate the information that is accumulated during the semantic analysis phase:

```
type app_kind = Tail_Call | Non_Tail_Call;;

type lexical_address =
  | Free
  | Param of int
  | Bound of int * int;;

type var' = Var' of string * lexical_address;;

type expr' =
  | ScmConst' of sexpr
  | ScmVarGet' of var'
  | ScmIf' of expr' * expr' * expr'
  | ScmSeq' of expr' list
  | ScmOr' of expr' list
  | ScmVarSet' of var' * expr'
  | ScmVarDef' of var' * expr'
  | ScmBox' of var'
  | ScmBoxGet' of var'
  | ScmBoxSet' of var' * expr'
  | ScmLambda' of string list * lambda_kind * expr'
  | ScmApplic' of expr' * expr' list * app_kind;;
```

As you can see, the type `var` has been replaced with `var'`, which now has room for an object of type `lexical_address`, which now identifies both the kind of variable, as well as its lexical address.

The output of the semantic analysis phase of your compiler should be of type `expr'`, and this is what the code generator (in the final project) shall receive.

All the work on the semantic analyzer should be placed in the `Semantics` module. You shall implement the functions:

- `annotate_lexical_addresses`
- `annotate_tail_calls`

- `auto_box`

You will find a function `semantics : expr -> expr'` that is already implemented, and that composes all the above functions in the correct order.

4.2 Boxing of variables

In class, we presented two criteria for boxing variables:

- The variable has (at least) one *read* occurrence within some closure, and (at least) one *write* occurrence in *another* closure.
- Both occurrences and neither both parameters, nor already refer to the same rib in a lexical environment.

As we mentioned in class, these rules are always sufficient, but sometime unnecessary, i.e., there are some cases in which boxing is not logically required, but using these criteria will still lead to boxing. Consider the following example:

```
(define foo
  (lambda (x)
    (set! x 1)
    (lambda () x)))
```

Note that the order of evaluation of the expressions in the body of procedure `foo` is known statically: the body of the lambda expression contains an implicit sequence of two expressions, that are evaluated from first to last. This means that logically, `x` need not be boxed. However, since the two criteria presented in class do not consider evaluation order, using those rules would result in `x` being boxed.

4.3 Examples of the semantic analysis

```
utop[22]> let show str = Semantic_Analysis.semantics
(Tag_Parser.tag_parse (Reader.nt_sexpr str 0).found);;
val show : string -> expr' = <fun>

utop[23]> show "x";;
- : expr' = ScmVarGet' (Var' ("x", Free))

utop[24]> show "(lambda (x y) (+ x y))";;
- : expr' =
ScmLambda' (["x"; "y"], Simple,
 ScmApplic' (ScmVarGet' (Var' ("+", Free)),
  [ScmVarGet' (Var' ("x", Param 0)); ScmVarGet' (Var' ("y", Param 1))],
  Tail_Call))

utop[25]> show "(set! x 34)";;
- : expr' =
ScmVarSet' (Var' ("x", Free), ScmConst' (ScmNumber (ScmRational (34, 1))))

utop[26]> show "(let ((x 45)) (set! x 34))";;
```

```
- : expr' =
ScmApplic'
 (ScmLambda' (["x"], Simple,
   ScmVarSet' (Var' ("x", Param 0),
    ScmConst' (ScmNumber (ScmRational (34, 1))))),
 [ScmConst' (ScmNumber (ScmRational (45, 1)))], Non_Tail_Call)

utop[27]> show "(let ((x 45)) (let ((y 56)) (let ((z 78) (w 89)) (list x y z w))))";;
- : expr' =
ScmApplic'
 (ScmLambda' (["x"], Simple,
   ScmApplic'
    (ScmLambda' (["y"], Simple,
      ScmApplic'
       (ScmLambda' (["z"; "w"], Simple,
         ScmApplic' (ScmVarGet' (Var' ("list", Free)),
          [ScmVarGet' (Var' ("x", Bound (1, 0)));
           ScmVarGet' (Var' ("y", Bound (0, 0)));
           ScmVarGet' (Var' ("z", Param 0));
           ScmVarGet' (Var' ("w", Param 1))],
          Tail_Call)),
       [ScmConst' (ScmNumber (ScmRational (78, 1)));
        ScmConst' (ScmNumber (ScmRational (89, 1)))],
       Tail_Call)),
    [ScmConst' (ScmNumber (ScmRational (56, 1)))], Tail_Call)),
 [ScmConst' (ScmNumber (ScmRational (45, 1)))], Non_Tail_Call)

utop[28]> show "(if x y)";;
- : expr' =
ScmIf' (ScmVarGet' (Var' ("x", Free)), ScmVarGet' (Var' ("y", Free)),
 ScmConst' ScmVoid)

utop[29]> show "(let* ((x #f) (y 'moshe)) (if x y))";;
- : expr' =
ScmApplic'
 (ScmLambda' (["x"], Simple,
   ScmApplic'
    (ScmLambda' (["y"], Simple,
      ScmIf' (ScmVarGet' (Var' ("x", Bound (0, 0))),
       ScmVarGet' (Var' ("y", Param 0)), ScmConst' ScmVoid)),
    [ScmConst' (ScmSymbol "moshe")], Tail_Call)),
 [ScmConst' (ScmBoolean false)], Non_Tail_Call)

utop[30]> show "(lambda (x) (list (lambda () x) (lambda (y) (set! x y))))";;
- : expr' =
ScmLambda' (["x"], Simple,
 ScmSeq'
  [ScmVarSet' (Var' ("x", Param 0), ScmBox' (Var' ("x", Param 0)));
```

```
   ScmApplic' (ScmVarGet' (Var' ("list", Free)),
    [ScmLambda' ([], Simple, ScmBoxGet' (Var' ("x", Bound (0, 0))));
     ScmLambda' (["y"], Simple,
      ScmBoxSet' (Var' ("x", Bound (0, 0)), ScmVarGet' (Var' ("y", Param 0))))],
    Tail_Call)])

utop[31]> show "(lambda (x) (lambda (u) (u (lambda () x) (lambda (y) (set! x y)))))";;
- : expr' =
ScmLambda' (["x"], Simple,
 ScmLambda' (["u"], Simple,
  ScmApplic' (ScmVarGet' (Var' ("u", Param 0)),
   [ScmLambda' ([], Simple, ScmVarGet' (Var' ("x", Bound (1, 0))));
    ScmLambda' (["y"], Simple,
     ScmVarSet' (Var' ("x", Bound (1, 0)), ScmVarGet' (Var' ("y", Param 0))))],
   Tail_Call)))

utop[32]> show "\"the factorial of 5 = ~{ (fact 5) }\n\"";;
- : expr' =
ScmApplic' (ScmVarGet' (Var' ("string-append", Free)),
 [ScmConst' (ScmString "the factorial of 5 = ");
  ScmApplic' (ScmVarGet' (Var' ("format", Free)),
   [ScmConst' (ScmString "~a");
    ScmApplic' (ScmVarGet' (Var' ("fact", Free)),
     [ScmConst' (ScmNumber (ScmRational (5, 1)))], Non_Tail_Call)],
   Non_Tail_Call);
  ScmConst' (ScmString "\n")],
 Non_Tail_Call)
```

5 Final Words

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.