CS 496: Homework Assignment 3
Due: 24 February, 11:55pm

1 Assignment Policies

Collaboration Policy. Homework will be done individually: each student must hand in their own answers. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

2 Assignment

This assignment consists in implementing a series of extensions to the interpreter for for the language called LET that we saw in class. The concrete syntax of the extensions, the abstract syntax of the extensions and the parser that converts the concrete syntax into the abstract syntax is already provided for you (`scan&parse`). Your task is to complete the definition of the interpreter, that is, the function `value-of` so that it is capable of handling the new language features.

Before addressing the extensions, we briefly recall the concrete and abstract syntax of LET. The concrete syntax is given by the grammar in Fig. 1. Each line in this grammar is called a *production* of the grammar. We will be adding new productions to this grammar corresponding to the extensions of LET that we shall study. These shall be presented in Section 3.

Next we recall the abstract syntax of LET, as presented in class. We shall also be extending this syntax with new cases for the new language features that we shall add to LET.

```
(define-datatype program program?
  (a-program
    (exp1 expression?)))

(define-datatype expression expression?
```

<Program>	::=	<Expression>
<Expression>	::=	<Number>
<Expression>	::=	<Identifier>
<Expression>	::=	- (<Expression>, <Expression>)
<Expression>	::=	zero? (<Expression>)
<Expression>	::=	if <Expression>
		then <Expression> else <Expression>
<Expression>	::=	let <Identifier> = <Expression> in <Expression>

Figure 1: Concrete Syntax of LET

```
  (const-exp (num number?))
  (var-exp (var identifier?))
  (diff-exp
    (exp1 expression?)
    (exp2 expression?))
  (zero?-exp
    (exp1 expression?))
  (if-exp
    (exp1 expression?)
    (exp2 expression?)
    (exp3 expression?))
  (let-exp
    (var identifier?)
    (exp1 expression?)
    (body expression?))
  )
```

3 Extensions to LET

This section lists the extensions to LET that you have to implement. This must be achieved by completing the stub, namely by completing the implementation of the function `value-of` in the file `interp.scm` of the supporting files.

3.1 Minus

Extend the interpreter to be able to handle a `minus` operator. For example,

```
> (run "-(minus(5),6)")
(num-val -11)
```

The additional production to the concrete syntax is:

<Expression> ::= minus (<Expression>)

The abstract syntax node for this extension is as follows:

```
(define-datatype expression expression?
  ...
  (minus-exp
    (exp1 expression?))
  )
```

You are asked to extend the definition of `value-of` so that it is capable of handling these new forms of expressions. In particular, it should be able to handle the abstract syntax representation of `-(minus(5),6)` which is:

```
(a-program (diff-exp (minus-exp (const-exp 5)) (const-exp 6)))
```

Here is the stub for the interpreter:

```
(define value-of
   (lambda (exp env)
     (cases expression exp
       (const-exp (num) (num-val num))

       ...

       (minus-exp (exp1)   (write "implement me!"))
  )))
```

3.2 Arithmetic Expressions

Extend the interpreter to be able to handle expressions such as `+(2,x)`, `*(3,4)` and `(3,5)`.

The additional productions to the concrete syntax are:

<Expression>	::=	+ (<Expression>, <Expression>)
<Expression>	::=	* (<Expression>, <Expression>)
<Expression>	::=	/ (<Expression>, <Expression>)

The abstract syntax node for this extension is as follows:

```
(define-datatype expression expression?
  ...
  (add-exp
    (exp1 expression?)
    (exp2 expression?))
  (mult-exp
    (exp1 expression?)
    (exp2 expression?))
  (div-exp
    (exp1 expression?)
    (exp2 expression?))
  )
```

Here is the stub for the interpreter:

```
(define value-of
   (lambda (exp env)
     (cases expression exp
       (const-exp (num) (num-val num))

       ...

       (add-exp (e1 e2) (write "implement me!"))

       (mult-exp (e1 e2) (write "implement me!"))

       (div-exp (e1 e2) (write "implement me!"))
  )))
```

3.3 Lists

Extend the interpreter to be able to handle operators `emptylist`, `cons`, `car`, `cdr` and `null?` with the same meaning they have in Scheme. Note that in order to implement these extensions, the set of *expressed values* must be extended accordingly. It now becomes:

$$\text{ExpVal} = \text{Int} + \text{Bool} + \text{List of ExpVal}$$

The corresponding implementation of expressed values in Racket is:

```
(define-datatype expval expval?
  (num-val
    (value number?))
  (bool-val
    (boolean boolean?))
  (list-val
    (list (listof expval?)))
  )
```

The additional production to the concrete syntax is:

<Expression>	::=	emptylist
<Expression>	::=	car (<Expression>)
<Expression>	::=	cdr (<Expression>)
<Expression>	::=	null? (<Expression>)
<Expression>	::=	cons (<Expression>, <Expression>)

For example,

```
> (run "cons(1,emptylist)")
(list-val (list (num-val 1)))

> (run "cons(cons(1,emptylist),emptylist)")
(list-val (list (list-val (list (num-val 1)))))

```

```
> (run "let x = 4
      in cons(x,
              cons(cons(-(x,1),
                        emptylist),
                   emptylist))")
(list-val (list (num-val 4) (list-val (list (num-val 3)))))

> (run "null?(emptylist)" )
(bool-val #t)

> (run "null?(cdr(cons(cons(1,emptylist),emptylist)))")
(bool-val #t)

> (run "cdr(cons(cons(1,emptylist),emptylist))")
(list-val '())

> (run "cons(cons(1,emptylist),emptylist)")
(list-val (list (list-val (list (num-val 1)))))
```

The abstract syntax node for this extension is as follows:

```
(define-datatype expression expression?
  ...
  (cons-exp
    (exp1 expression?)
    (exp2 expression?))
  (car-exp
    (exp1 expression?)
    (exp2 expression?))
  (cdr-exp
    (exp1 expression?)
    (exp2 expression?))
  (null-exp
    (exp1 expression?))
  (emptylist-exp ()))
```

Here is the stub for the interpreter:

```
(define value-of
   (lambda (exp env)
     (cases expression exp
       (const-exp (num) (num-val num))

       ...

       (cons-exp (e1 e2) (write "implement me!"))

       (car-exp (e1) (write "implement me!"))

       (cdr-exp (e1) (write "implement me!"))

```

```
      (null-exp (e1) (write "implement me!"))

      (emptylist-exp () (write "implement me!"))
 )))
```

4 Submission instructions

Submit a file named `HW3_<SURNAME>.zip` through Canvas. Include only the supporting files uploaded into Canvas but where `interp.scm` has been completed, as described in this document. Please write your name in the source code using comments. Your grade will be determined as follows, for a total of 100 points:

Section	Grade
3.1	20
3.2	30
3.3	50