

## Introduction

In this lab we take a tour of the LET language's interpreter and walk through the process of making a simple extension to it.

## Setup

Retrieve the file <http://www.cis.syr.edu/courses/cis352/code/let.zip> and unzip it. It will create a directory, `let-lang`, with seven `.scm` files for this lab and the next homework. Alternatively, create a `let-lang` directory, visit [this directory](#) and copy each of the files into your `let-lang` directory. These files are:

File	Contents
<code>data-structures.scm</code>	Data-type representations of LET's <i>values</i> and of the variable environment
<code>drscheme-init.scm</code>	Compatibility code for DrScheme ( <i>Do not modify!!</i> )
<code>environments.scm</code>	An interface for the environment data type and the initial environment: $[i \mapsto 1, v \mapsto 5, x \mapsto 10]$ .
<code>interp.scm</code>	The interpreter of LET
<code>lang.scm</code>	The syntactic spec of LET
<code>tests.scm</code>	Tests of LET interpreter
<code>top.scm</code>	The top level module for the interpreter. Loading this file also loads everything else.

In this lab and the next homework you will be modifying each of these files *except for* `drscheme-init.scm` and `top.scm`. Please leave these two alone, *especially* `drscheme-init.scm`!

## Running Some LET Programs

Load `top.scm` into DrScheme and hit the Run button. Try evaluating:

- `(run "12")`
- `(run "x")`
- `(run "v")`
- `(run "foo")` *[runtime error]*
- `(run "-(x,v)")`
- `(run "zero?(0)")`
- `(run "if zero?(v) then -(x,3) else v")`
- `(run "let c = 100 in -(c,x)")`
- `(run "let c = 100; m = 1000 in +(m,c)")` *[parsing error]*

In general, if *prog* is a string consisting of a syntactically correct LET program, then `(run prog)`: (i) parses *prog*, (ii) runs the LET program, and (iii) reports the result. If *prog* has syntax errors, you get a cryptic error message from the parser—as in the last example above. A run-time error, as in the `(run "foo")` example, occurs when the program is syntactically correct, but something goes wrong when the interpreter tries to evaluate the program (e.g., the variable `foo` is not in the environment).

Also try evaluating:

- `(run-all)`

This runs all the tests in `tests.scm` and gives you a report on what worked and what didn't. The run procedure is good for one-time experiments; `run-all` is an easy way of running a bunch of tests on the interpreter. As part of the development of the interpreter you will add tests to the test harness to gather evidence that your interpreter is working as advertised.

## Adding to the Test Harness

Now load `tests.scm` into DrScheme. You will see that there is only one thing being defined, `test-list`, which is a list consisting of sublists of the form:

`(test-name program expected-result)`

For example, the report from the test `(positive-const "11" 11)` is:

```
test: positive-const
correct
```

meaning: *in the test positive-const the program resulted in the expected value.* Let us vandalize `tests.scm` by changing this test to `(positive-const "11" 10)`. (Be sure to save the changed `tests.scm` and reload `top.scm`.) Now run: `(run-all)`. Now the test reports:

```
test: positive-const
11
correct outcome: 10
actual outcome: #(struct:num-val 11)
incorrect
```

Also, at the very end you get the note:

`incorrect answers on tests: (positive-const)`

Now, undo our vandalism in `tests.scm`, save the file, and reload `top.scm` yet again.

If you look further down in `tests.scm`, you will see

`(test-unbound-var-1 "foo" error)`

meaning: *evaluating foo should result in a run-time error.* So, if evaluating `foo` *does* result in a run-time

error, then the test is passed. A parsing error is also reported as a failed test.

At the very end of the lists of tests, but before the final three end-parens, add:

```
;; Lab 5 tests
(lab5-1
  "if zero?(-(x,x)) then v else i" 5)
```

Save `tests.scm`, reload `top.scm`, and run: (`run-all`). The last test report should read:

```
test: lab5-1
correct
```

Now you know how to add tests to the test harness.

## Extending the LET Language

Now let us work through the steps of doing EOPL Exercise 3.6 (page 72), which reads

Extend the language by adding a new operator `minus` that takes one argument,  $n$ , and returns  $-n$ . For example, the value of `minus(-(minus(5),9))` should be 14.

To incorporate this new operator, we need to modify LET's: (1) grammar (`lang.scm`), (2) interpreter (`interp.scm`), and (3) test suite (`tests.scm`).

**1. Modifying the grammar** First open `lang.scm` and have a look around. Most of the file consists of the definition of two lists: `the-lexical-spec` and `the-grammar`.

The `the-lexical-spec` list explains how to translate a string of characters to a list of grammatical tokens. (Think: *letters/sounds*  $\mapsto$  *words*.) This translation process is called *scanning*; see EOPL Section

B.1 (pages 279–282) for details. To see this translation in action, try:

- `(just-scan "2")`
- `(just-scan "foo")`
- `(just-scan "-(x,-(v,2))")`
- `(just-scan "if zero?(foo) then v else i")`

We **will not** be modifying the `the-lexical-spec`.

The `the-grammar` list, explains how to translate a list of tokens to a syntax tree (i.e., a data type representation of the program as a grammatical construct). (Think: *words*  $\mapsto$  *sentences*.) This translation process is called *parsing*; see EOPL Section B.2 (pages 282–283) for details. To see this translation in action, try:

- `(scan&parse "2")`
- `(scan&parse "foo")`
- `(scan&parse "-(x,-(v,2))")`
- `(scan&parse "if zero?(foo) then v else i")`

The elements of `the-grammar` are grammar rules called *productions*. The general form of a production is: *(syntactic-category (grammatical-stuff\*) production-name)* Here are the productions for LET:

```
(program (expression) a-program)
(expression (number) const-exp)
(expression (identifier) var-exp)
(expression
  ("zero?" "(" expression ")")
  zero?-exp)
(expression
  ("-" "(" expression "," expression ")")
  diff-exp)
(expression
  ("if" expression "then" expression
   "else" expression)
  if-exp)
(expression
```

```
("let" identifier "=" expression "in"
                                     expression)
let-exp)
```

*Note:* `number` and `identifier` are predefined syntactic categories. Here is a BNF version of the above.

```
<program> ::= <expression>
<expression> ::= <number>
                | <identifier>
                | zero?(<expression>)
                | - (<expression> , <expression>)
                | if <expression> then <expression> else <expression>
                | let <identifier> = <expression> in <expression>
```

See EOPL Section B.3 for the rules for formulating productions; we shall talk about these in more detail in class.

Now, back to our problem; `minus` is a one-argument operator just like `zero?`. Let us pretend we know what we are doing and copy the `zero?` production, changing each `zero?` to `minus`. That is, add

```
(expression ;; Lab 5
  ("minus" "(" expression ")")
  minus-exp)
```

to the-grammar's list of productions, then try:

- `(scan&parse "minus(2)")`
- `(scan&parse "-(minus(x),v)")`

**2. Modifying the interpreter** Open `interp.scm` and have a look around. Most of the file consists of the definition of the procedure `value-of` which is a big cases expression based on production names. Since `minus` is an one-argument operator like `zero?`, it is reasonable (and correct) to guess that the `minus`-case should look like:

```
(minus-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    ???))
```

When this case holds, we are evaluating an expression of the form `minus(stuff)` and `exp1` is the syntax tree corresponding to `stuff`. Then `val1` gets the value of `exp1`, which should be a number which we need to negate and return — except for the fact that there is another layer of data types to deal with.

Open up `data-structures.scm` and look at the `expval` data type. LET has two sorts of *values*, integers and booleans, and these values are always packaged in a `expval`-data-type. Try:

```
➤ (define n (num-val 12))
➤ n
➤ (define b (bool-val #f))
➤ b
```

To extract the integer hiding in `n`, we use the `expval->num` procedure. Try:

```
➤ (expval->num n)
➤ (expval->bool b)
```

So the `minus-exp` case becomes:

```
(minus-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((num1 (expval->num val1)))
      ???))
```

Hence, `num1` is the integer value we have to negate and return. But to return this new integer value, it has to be repackaged in an `expval` data type. Thus the final form of the `minus-exp` case is:

```
(minus-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((num1 (expval->num val1)))
      (num-val (- num1))))))
```

You can see this unpacking and repacking going on in the `diff-exp` and `zero?` cases.

Save your files, reload `top.scm` and try:

```
➤ (run "minus(12)")
➤ (run "minus(-(minus(9),3))")
```

**Some words of warning.** Lots of bugs in interpreter code stem from:

- Forgetting to package a value (e.g., integer, boolean). For example, in place of the last line of the `minus-exp` code, we could goof by writing: `(- num)`.
- Confusing LET syntax (to be evaluated) and LET values (to be computed over). For example, in place of lines 2 and 3 of the `minus-exp` code, we could goof by writing simply:

```
(let ((num1 (expval->num exp1))) ...
```

Scheme's dynamic typing can make mistakes like the above hard to track down.

**3. Modifying the test suite** We already know how to do this by adding tests to `tests.scm`. Add tests to check that

1. `minus(12)` evaluates to `-12`
2. `minus(-(minus(9),3))` evaluates to `12`

## Your Problems

Do EOPL Exercise 3.7 (page 72), which reads

Extend the language by adding operators for addition, multiplication, and integer quotient.

Be sure to:

1. Add comments to the sources to make what you changed and what you added.
2. Add tests to the test harness to exercise your extensions.
3. Do a `(run-all)` at the end to make sure everything works.

### What to Hand In

- The source code of the files that you changed.
- The output of your final `(run-all)`.