POPL Practicals

Alan Wood

2014

Contents

Part I

Principles

Segment 1

Scheme

This first segment will introduce you to the `scheme` language, which will be used as the 'defining' or 'meta' language for the 'defined' or 'object' languages (see the lectures!) that you will be creating and using to investigate the various principles of programming langauges in the rest of POPL Part 1.

The practical exercises are as much a part of POPL as the lectures so you must expect to invest a significant learning effort in them. They *should* take a *lot* of your time! It is very unlikely that you will you get through all of the exercises during the time-tabled practical sessions since you'll be spending quite a lot of time reading, questioning, backtracking, debugging, re-designing etc. So expect to do most of the meaty exercises in your own time.

This segment, as in all the following ones, is organised as a sequence of exercises: some are paper-and-pencil, others involve programming. In all the segments, exercises with headings in solid box are *essential* — you should complete all of these. *Optional* exercises are indicated by a dashed box — try to do these, but they are not *essential*.

Working Method

I *very strongly* recommend the following organisation of your work in the POPL practicals:

1. Create new directories for each segment, possibly naming them by the segment number.

2. Within a segment's directory, create a separate directory for each major 'unit' (section, set of exercises etc.).

3. Create all the files for a particular unit within its directory, even if this means making duplicate copies of existing files. You will sometimes be given 'boiler-plate' code,[1] or test data etc. You should *copy* that into the exercise's directory before starting.

If you want to follow some other organisation, it's your choice, but you may get tangled up with the wrong versions of files (from earlier exercises, for instance). As I said, I *very* strongly recommend this way of doing things!

0. Running `scheme`

For the practicals in POPL Part 1 you will be using the *DrRacket* `scheme` IDE running under Linux or Windows: your choice, but I shall assume the Linux implementation in the following.

[1] Program text that is required, but doesn't add to the learning purpose of the exercise.

You should work in some convenient directory in your own file space — I shall refer to it as your 'working directory'.

Exercise 0 Depending on whether you're using Linux or Windows, do the following:

Linux	Windows
• Open a terminal window and `cd` to your working directory. • Issue the `drracket&` command to start *DrRacket* in the background — that's what the `&` is for you'll recall! After that you can ignore the terminal window until *DrRacket* exits.	• Start DrRacket from the Start button

The GUI has two windows: the lower ('interactions') window evaluates any `scheme` expression that you type in. The upper ('definitions') window is for typing and editing definitions. All the expressions and definitions typed into the definitions window are executed when the Run button in the menu bar is clicked. When Run is clicked, the interactions window is reset ready for you to evaluate further expressions.

NB. The first time you use *DrRacket* you will need to tell it which dialect of scheme ('language') to use. Since, from Segment 2, we will be following the exercises in *Essentials of Programming Languages*, use the `eopl` dialect — this will ensure that all the extra tools used in *EOPL* are available. To set this as the default, hit `ctrl-L` and select the top radio button labelled 'Use the language declared in the source.', and then close the dialogue. This will cause the line:

```
#lang racket
```

to appear in the top window (if it is not already there!). Then press the Run button.[2]

I also recommend selecting the Open files in separate tabs (not separate windows) option in the Edit|Preferences... dialogue under the General tab.

You should experiment with the *DrRacket* system to familiarise yourself with its use, before moving on to the next section. For example, try the following:

- In the interactions window type:

```
> 123456
> (* 234 987654321)
> (define three 3)
> three
```

[2]You will be using the `racket scheme` dialect in Segment 1, but you will change to using the `eopl` dialect from Segment 2 onwards.

- In the definitions window there is no > prompt, as the expressions and definitions typed here are evaluated when Run is clicked. Try typing all the above expressions (without the >) into the definitions window and then Run it.

  Note that *all* the separate expressions are evaluated in sequence but their results are not seen[3] in the interactions window. In practice it is unusual to have any expression other than a definition in the definitions window.

Finally, the contents of the definitions window can be saved to file using the File|Save Definitions As... menu item. A previously saved file can be then reloaded into a new tab with Ctrl-O.
□

1.1 Learning `scheme`

There are two fundamental resources for learning `scheme`:

1. The Revised[5] Report on Scheme (R^nRS). Available on-line at:

   pdf: `www.schemers.org/Documents/Standards/R5RS/r5rs.pdf`

   html: `www.schemers.org/Documents/Standards/R5RS/HTML/`

2. *SICP* [**?**][4]— available free on-line at:

   `http://mitpress.mit.edu/sicp/full-text/book/book.html`

It is possible to learn enough `scheme` to start on the exercises by reading through the early parts of R^nRS, trying out some examples of your own devising. Back this up by questioning *us* and looking up information in *SICP*. In fact, I recommend this approach as 'bootstrapping' yourself into a new language is a *very* useful skill to acquire!

However, there are some on-line tutorials available which take a more sequentially structured approach. The disadvantage is that they might assume you're using particular implementations, system set-ups etc. which don't apply and so can be confusing. By all means try these though if you prefer. A couple of reasonable tutorials are at:

- `www.cs.hut.fi/Studies/T-93.210/schemetutorial/schemetutorial.html`
- `www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html`

1.1.1 Reading

From segment 2 onwards, you will be following the exercises in *Essentials of Programming Languages* extensively. There will also be suggested readings from that book. For *this* segment you should read [Ch. 1] which will help your understanding of `scheme`, provide a few extra

[3] This behaviour varies according to the language dialect chosen: `eopl` behaves as stated, whereas `racket` (the default) shows the values of *every* expression in the definitions window!

[4] Note on citations: those with *authors* listed refer to the Bibliography (page **??**). Those *without* authors, refer to *Essentials of Programming Languages* [**?**].

exercises for you to try, and introduce the techniques of 'data-oriented design' which is followed in the subsequent segments.

Exercise 1 You should be prepared to spend most of the first timetabled hour of this segment on this exercise! However, by all means move on to the programming exercises when you feel you can.

- Look at the learning resources — R^nRS, *SICP*, and the tutorials mentioned above — and decide which suits you best.
- Start learning `scheme`.

□

Background to the Programming Exercises

The following two sections are slightly modified versions[5] of the second and third practicals that were given to our first-year students for about a decade from the mid-90s.[6] They are therefore only moderately difficult, as the students hadn't had as much programming experience as our current Stage 2. So they shouldn't be too intellectually challenging for you with your greater experience and knowledge.

However, those students *had* had about 10 lectures which used `scheme`,[7] whereas you have had about an hour's experience of `scheme` (unless you've used it before) as the first part of this segment. So you will need to make use of your greater experience in programming in general to complete the exercises.

1.2 Sentencing

This section is concerned with helping your understanding of:

- lists,
- procedure definitions, and
- recursion.

as well as giving practice in *reading* programs written by others.[8]

You will build procedures which operate on *lists* — these lists will contain words which you will use to generate English sentences. You will not have to write the whole program (collection of procedures) to generate these sentences, but you will have to *understand* the whole program since you'll need to define some of the procedures which are (deliberately) missing.

[5] Because of the heritage of the exercise sections, it's possible that there are still some oddities in the text due to inaccurate editing. I'd be grateful for you to let me know when you find them, and I'll correct them for next time ... none were pointed out to me last year!

[6] Which is turn were slightly modified versions of material written by MIT (© Massachusetts Institute of Technology, 1988).

[7] And the course text was Abelson and Sussman!

[8] This is a *vitally* important skill.

1.2.1 Generating Sentences

Background

We want a program that will generate English sentences. In order to approach this vastly complex topic, we will restrict ourselves to considering only very simple sentences — for our purposes a sentence will consist of the word "the" followed by a *noun phrase* followed by a *verb phrase*.

A noun phrase will consist of either a *noun*, or an *adjective* followed by a noun phrase.[9] A verb phrase is either a *verb*, or a verb followed by an *adverb*.[10] Make sure that you understand these two definitions — the program will be designed by following their structure.

We will represent words as Scheme *symbols*, and the words available to the system will be stored in lists, as follows:

```
(define noun-list (list 'dog 'cat 'student 'professor 'book 'computer))

(define verb-list (list 'ran 'ate 'slept 'drank 'exploded 'decomposed))

(define adjective-list (list 'red 'slow 'dead 'pungent 'over-paid 'drunk))

(define adverb-list (list 'quickly 'slowly 'wickedly 'majestically))
```

Phrases will be represented as lists of words (that is, lists of symbols). For example `(ate)` and `(ate quickly)` are verb phrases; `(professor)` and `(dead pungent student)` are noun phrases.[11]

Exercises

You need the scheme code for the four word lists defined above, so start by typing those into your definitions window.[12] Then click Run to 'install' them. If there are any error messages at this point go back and correct your definitions. If not, test them by *evaluating* them — typing the defined names into the *interaction* window. For example, the value of `noun-list` is:

```
(dog cat student professor book computer)
```

Don't forget to save your definitions window periodically!

Note. You will need to fill in 'gaps' in the some of the following definitions. These are all marked by `???`.

The top-level procedure is the one that generates sentences. It takes no arguments, since it should produce a new sentence each time that it is evaluated. As stated in subsection 1.2.1, `(sentence)` should generate a list consisting of the word `the` followed by a noun phrase followed by a verb phrase, so we assume that two procedures `noun-phrase` and `verb-phrase` exist, and that they produce lists corresponding to the two phrase types. This means that we will need to join several lists together to form a single list. We shall use `append` for this:

[9] Notice that this is a *recursive* definition — noun phrase is defined in terms of itself.

[10] On the other hand, this is *not* recursive.

[11] The latter is a noun phrase since it consists of an adjective `dead`, followed by a noun phrase (an adjective `pungent`, followed by a noun phrase (a noun `student`)).

[12] Copy and paste is OK, too!

```
(define (sentence)
  (append (append ??? (noun-phrase)) (verb-phrase))
)
```

Exercise 2 Work out what should replace the `???` in this definition, and edit it into your program file. You can't test this yet, as we need to define the other procedures.
□

Since the words which will make up the various phrases are to be picked at random from the word lists, we need a procedure to do just this. This involves using the built-in procedure `(random n)` which evaluates to a random integer between 0 and `n`−1. We will also need to use the built-in procedure `(list-ref lst n)` which evaluates to the $\texttt{n}^{\text{th}}$ element (counting from 0) of the list `lst`.

Exercise 3 Complete the following, type it in to your program, and test it (work out *how* to test it yourself):

```
(define (pick-random lst)
  (list-ref lst (random ??? ))
)
```

□

Exercise 4 Write a procedure `a-noun` (that takes no parameters) that selects a noun from `noun-list` at random. Similarly, write procedures `a-verb`, `an-adverb`, and `an-adjective`.
Test all of these procedures.
□

In order to generate phrases, we will write a procedure called `either` which takes as arguments two *procedures* (which take no parameters) and evaluates one or the other of them. The definition that we shall use is:

```
(define (either a b)
  (if  (= (random 2) 0)  (a) (b) )
)
```

Check that you understand this procedure,[13] and include it in your program.

Now you have all the tools you need to construct the procedures `noun-phrase` and `verb-phrase` Using `either`, the definition of these two procedures is a matter of translating the definitions of *noun phrase* and *verb phrase* given in subsection 1.2.1 into scheme notation.

Exercise 5 Design and test `noun-phrase` and `verb-phrase` as defined above.
□

For reference, the Appendix (page 51) lists all the code that you need to write (without the 'gaps' being filled, of course!).

[13] You probably don't! There's a gotcha here!

1.2.2 Going Further

These suggestions can be done in any order. However, it would be worth *thinking* about them all even if you don't have time to implement them.

Why is `(either ...)` so weird?

It would seem at first sight that `either` has been made overly complicated by using procedural-valued parameters. A simpler (but incorrect) definition might be:

```
(define (either a b)
  (if (= (random 2) 0) a b)
)
```

along with appropriate modifications to the rest of the program.

In fact, this is fine for its use in the `verb-phrase` procedure — re-define `verb-phrase` appropriately, and confirm that this is the case. However, it does *not* work with `noun-phrase`.

Exercise 6 What happens if you re-define `noun-phrase` to use the new version of `either`? Why?
□

Tidying-up "the"

You might feel that the way we've handled "the" is a hack — surely it should be part of the definition of *noun phrase*, rather than just being tacked on to the front when constructing sentences? If we could define *noun phrase* differently, then `sentence` could be:

```
(define (sentence) (append (noun-phrase) (verb-phrase)))
```

which is much clearer.

Exercise 7 Unfortunately, the näive approach — redefine *noun phrase* to be "the" followed by a noun, or "the" followed by an adjective followed by a noun phrase — doesn't work. Why?[14]
□

Exercise 8 Supply a correctly re-defined `noun-phrase` which allows the new definition of `sentence` to generate the same sentences as the old version.
□

The sentences produced all use *intransitive* verbs, that is there is only a single noun — the subject. It would be much more interesting if we could also generate sentences using *transitive* verbs which can then contain an *object* — a second noun. In fact, once we have

[14] Give an example of an undesirable phrase that could be generated.

transitive verbs, the object can be a noun *phrase*, so that we could generate a sentence such as `(the dead drunk professor crushed wickedly the over-paid red computer)`

Exercise 9 Augment the sentence generator to produce such sentences. You will need to classify verbs as transitive or intransitive (using different lists would seem the easiest). □

Incidentally, there are verbs which are neither transitive nor intransitive — *to be* is an example.[15]

1.3 Caring

This section helps to consolidate your understanding of:

- Program reading and modifying,
- Working with Lists

There is a *lot* of program reading to do here *before* the first programming exercise. Please don't skip through this too quickly . . . it is essential that you understand the material before trying the exercises. In particular, you should slow down your reading speed when looking at the `scheme` code: make sure you *really* understand every element of a procedure before moving on.

1.3.1 Background

These exercises deal with a drastically simplified version of a famous program called `doctor` that was developed at MIT in the early 1960s by Professor Joseph Weizenbaum.[16] The `doctor` program engages in a "conversation" with the computer user, in which it mimics a nondirective style of psychotherapy. In the following sample conversation, the user's responses are preceded by the prompt symbol `**`, and the program's responses are shown in upper case. In order to simplify the program, we omit most punctuation and represent the responses of both the user and the machine as Scheme lists.

```
> (visit-doctor 'Bill)
(HELLO, BILL)
(WHAT SEEMS TO BE THE TROUBLE?)

**(everyone hates me)
(WHY DO YOU SAY EVERYONE HATES YOU)

**(whenever i enter a room no one will look me in the eye)
(MANY PEOPLE HAVE THE SAME SORTS OF FEELINGS)

**(they laugh at me behind my back)
(YOU SEEM TO THINK THEY LAUGH AT YOU BEHIND YOUR BACK)
```

[15] It is a *copulative* verb `:-)`

[16] The more famous name for this program is 'Eliza' [**?**].

```
**(they get together and talk about how they hate me)
(MANY OF MY PATIENTS HAVE TOLD ME THE SAME THING)

**(even my cat hates me)
(YOU FEEL THAT EVEN YOUR CAT HATES YOU)

**(when i was a teenager she said i was ugly and stupid)
(PLEASE CONTINUE)

**(she bit me all the time)
(WHY DO YOU BELIEVE SHE BIT YOU ALL THE TIME)
```

Although the `doctor` program seems to understand and reply to the user's remarks, in fact the program has just two simple methods for generating a response. The first method is to take the user's reply, change some first-person words like "I," "me," "my," and "am" to the corresponding second-person words, and append the transformed response to a qualifying phrase such as "why do you say" or "you seem to think". The second method is to ignore what the user types and simply respond with some sort of hedge like "please continue" or "many people have the same sorts of feelings." The program chooses one of these two methods at random.

Every interactive program, including the Scheme interpreter itself, has a distinguished procedure called a *driver loop*. A driver loop repeatedly accepts input, determines how to process that input, and produces the output. The `visit-doctor` procedure first greets the user, then asks an initial question and starts the driver loop.[17]

```
(define (visit-doctor name)
  (begin
    (print (list 'hello name))
    (print '(what seems to be the trouble?))
    (doctor-driver-loop name)))
```

The driver loop prints a prompt and reads in the user's response. If the user says `(goodbye)`, then the program terminates. Otherwise, it generates a reply according to one of the two methods described above and prints it.[18]

```
(define (doctor-driver-loop name)
  (begin
    (newline)
    (display '**)
```

[17] This procedure, and several others in the section, use the special form `(begin expr1 expr2 ... exprN)`, which takes a series of expressions (`expr1` ... `exprN`) evaluates them all *in begin*, and returns the value of the last (`exprN`). The values of all but the last expression are 'thrown away'. This might seem rather silly! However, it is used when you wish to execute all but the last expression for their *side-effects*, such as (as in this case) printing a value. This is commonly needed when dealing with *input* and *output* in programs. See *SICP* §3.1.1 [**?**].

[18] This procedure, and several of the others, use the predicate `equal?`. This is the most general way of testing whether two values are 'the same', and works for all types of value (not just numbers which is the case for the `=` predicate). See R^nRS, or *SICP* for details.

```
    (let ((user-response (read)))
      (if (equal? user-response '(goodbye))
                (begin
                   (print (list 'goodbye name))
                   (print '(see you next week))
                )
                (begin
                    (print (reply user-response))
                    (doctor-driver-loop name)
                )))))

(define (reply user-response)
  (if (fifty-fifty)
        (append (qualifier) (change-person user-response))
        (hedge)))
```

The predicate `fifty-fifty` used in `reply` is a procedure that returns true or false with equal probability.

```
(define (fifty-fifty) (= (random 2) 0))
```

Qualifiers and hedging remarks are generated by selecting items at random from appropriate lists:

```
(define (qualifier)
  (pick-random '((you seem to think)
                 (you feel that)
                 (why do you believe)
                 (why do you say))))

(define (hedge)
  (pick-random
   '((please go on)
     (many people have the same sorts of feelings)
     (many of my patients have told me the same thing)
     (please continue))))
```

The basis for the procedure that changes selected first person words to second person is the following `replace` procedure, which changes all occurrences of a given `pattern` in a list `lst` to a `replacement`:

```
(define (replace pattern replacement lst)
  (cond ((null? lst) '())
        ((equal? (car lst) pattern)
                 (cons replacement
                       (replace pattern replacement (cdr lst)))
        )
        (else (cons (car lst)
                    (replace pattern replacement (cdr lst)))
        )))
```

This is used to define a procedure `many-replace`, which takes as inputs a list `lst` together with a list of `replacement-pairs` of the form:

((*pat1 rep1*) (*pat2 rep2*) ...)

It replaces in `lst` all occurrences of *pat1* by *rep1*, *pat2* by *rep2*, and so on.

```
(define (many-replace replacement-pairs lst)
  (if (null? replacement-pairs)
        lst
        (let ((pat-rep (car replacement-pairs)))
               (replace (car pat-rep)
                        (cadr pat-rep)
                        (many-replace (cdr replacement-pairs)
                                      lst)))))
```

Changing the selected words is accomplished by an appropriate call to `many-replace`:

```
(define (change-person phrase)
  (many-replace '((i you) (me you) (am are) (my your))
                phrase))
```

The procedure `pick-random` used by `qualifier` and `hedge` picks an element at random from a given list:

```
(define (pick-random lst) (list-ref lst (random (length lst))))
```

Figure 1.1 shows the pattern of procedure calls indicated in the text of the `doctor` program.

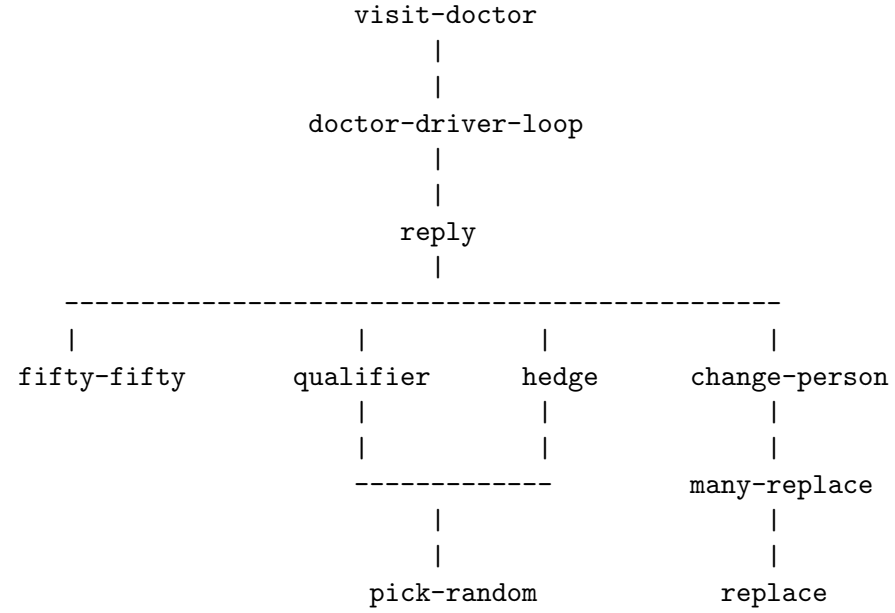

Figure 1.1: Procedure calls in the `doctor` program.

1.3.2 Exercises

In the following exercises, you will be making modifications to the procedures listed above, as well as adding new ones to the program. Take a copy of:

```
/shared/rentedfs/cs-course/popl/Resources/Practicals/CurrentYear/caring.scm
```

into your working directory. Open it in *DrRacket* — use the File — Open menu item.

Exercise 10

- Run through a brief session with the modified program — it's not very clever . . . yet!
- Edit the `qualifier` and `hedge` procedures to increase the doctor's repertoire of qualifying and hedging phrases.
- Test your modified program.

□

What is the result of evaluating

```
(change-person '(you are not being very helpful to me))
```

We can improve the `doctor` program by having it not only change first person words to second person, but also second person to first. For instance, if the user types

```
(you are not being very helpful to me)
```

the program should respond with something like

```
(YOU FEEL THAT I AM NOT BEING VERY HELPFUL TO YOU)
```

Thus, "are" should be replaced by "am," "you" by "i," "your" by "my," and so on. (We will ignore the problem of having the program decide whether "you" should be replaced by "i" or by "me.")

Exercise 11 One idea for accomplishing this replacement is to simply add the pairs `(are am)`, `(you i)`, and `(your my)` to the list of pairs in the `change-person` procedure. Edit the procedure to do this. Now try evaluating

```
(change-person '(you are not being very helpful to me))
```

What does the modified procedure return? Does it matter whether you add the new pairs to the beginning or the end of the replacement list?
□

Exercise 12 Think[19] carefully how to describe the bug in the method of implementing replacement used above.
□

Exercise 13 Design a correct replacement method that will accomplish both kinds of replacement (first person by second person as well as second person by first person). Write,

[19] Thinking is a practical exercise as well as programming!

test, and debug procedures that implement your replacement strategy. Install these in the `doctor` program.
□

Another improvement to the `doctor` program is to give it a third method of generating responses. If the doctor remembered everything the user said, then it could make remarks such as

```
(EARLIER YOU SAID THAT EVERYONE HATES YOU)
```

Exercise 14 Add this method to the program as follows.

1. Modify the program so that `doctor-driver-loop` maintains a list of all user responses.[20]
2. Modify the program so that `reply` occasionally replies by picking a previous user response at random, changing person in that response, and prefixing the modified response with "earlier you said that." If you want more control over how often the program uses each response method, you can use the following predicate, which returns true `n1` times out of every `n2`:

   ```
   (define (prob n1 n2) (< (random n2) n1))
   ```

□

Exercise 15 The doctor currently sees only one patient, whose name is given in the call to `visit-doctor`. When that patient says `(goodbye)`, `visit-doctor` returns to the Scheme interpreter. Modify the program so that the doctor automatically sees a new patient after the old one goes away, and provide some way to tell the doctor when to stop. For example, `visit-doctor` might terminate after seeing a particular number of patients (supplied as an argument) or when it sees a patient with some special name (such as `AlanSugar`). You may use the following procedure to find out each patient's name:

```
(define (ask-patient-name)
  (print '(next!))
  (print '(who are you?))
  (car (read)))
```

Now a session with the doctor might look like

```
> (visit-doctor)
(NEXT!)
(WHO ARE YOU?) (Alan Burns)
(HELLO, ALAN)
(WHAT SEEMS TO BE THE TROUBLE?)

**(everyone taking POPL hates me)
```

[20] For people who have `scheme` experience, or have been reading *SICP*, do not use `set!` in order to implement this — it isn't necessary.

```
(WHY DO YOU SAY EVERYONE TAKING POPL HATES YOU)
...

**(goodbye)
(GOODBYE, ALAN)
(SEE YOU NEXT WEEK)
(NEXT!)
(WHO ARE YOU?) (Bruce Forsyth)
(HELLO, BRUCE)
(WHAT SEEMS TO BE THE TROUBLE?)
...
**(goodbye)
(GOODBYE, BRUCE)
(SEE YOU NEXT WEEK)
(NEXT!)
(WHO ARE YOU?) (AlanSugar)
(TIME TO GO HOME)
>
```

□

1.3.3 Going Further

Exercise 16 (Open-ended design project) Design and implement another improvement that extends the capabilities of the `doctor` program in some significant way. For example, you could give the program the ability to make a response that relates to what the user said. The response to "I am often depressed" could be "When you feel depressed, have a pizza at Caesars." You can implement this by associating canned responses with key words, so that when the user mentions one of the key words, the program selects one of the responses associated with that word. The keyword-response list could look like

```
( ((depressed suicide)
     ( (when you feel depressed, have a pizza at Caesars)
       (depression is a disease that can be treated) )
  )
  ((mother father parents)
     ( (tell me more about your family)
       (why do you feel that way about your parents?) )
  )
)
```

The data structure used here is a list of lists, each containing a list of keywords and a list of responses. A variation on this trick is to have the "canned" responses include slots to be filled in with the key word that triggered the response. For example, with the following keyword-response lists the doctor might respond to a sentence including "father" with "Tell me more about your father."

```
( ((depressed suicide)
     ( (when you feel depressed, have a pizza at Caesars)
       (depression is a disease that can be treated) )
  )
  ((mother father parents)
     ( (tell me more about your *)
       (why do you feel that way about your * ?) )
  )
)
```

Alternatively, you could generalise the structure of the `doctor` program so that instead of having a collection of keywords to check for, the program has a data structure containing a collection of arbitrary predicates (procedures) and associated "response procedures". If the user's typed response is found to satisfy one of the predicates, then the program uses one of the associated response procedures to generate its reply. For instance, including the predicate

```
(lambda (user-response) (< (length user-response) 3))
```

would allow the program to react to very short answers with a reply such as "Could you say more?"

Implement your modification. (You need not feel constrained to follow the suggestions given above.)
□

Segment 2

Interpreters: Tools and Techniques

Background

All the subsequent segments assume that you have *continuous* access to a copy of the 3^{rd} edition of *Essentials of Programming Languages* [**?**]. Although many of the exercises here will be taken directly from *EOPL*, they will often refer to the *text* of that book, which you will need to read.

Some exercises will be in addition to, or be variations on the *EOPL* exercises. However, to help orient yourself, the *EOPL* exercise corresponding to each of those here is noted in the margins.

The amount of detail given in these segments will decrease as you go along. You will *have to* refer to *EOPL* for many of the later exercises as they will not be repeated here. However, by the time you get there you will have a sound knowledge of the methods for producing the solutions, and will be able to concentrate on actually solving the questions.

Introduction

In order to become familiar with the tools, you'll implement an interpreter for a very simple language. This will follow *EOPL* quite closely, although the material will be presented with a view to learning the tools rather than the language itself.

Reading

a) If you've not read [Chapter 1], do so.

b) The following are essential for this (and following) segments:

[§2.4] Describes the `(define-datayape ...)` and corresponding `(cases ...)` special forms. These are implemented as *macros* and so have a non-standard evaluation model. These will be used *extensively* in implementing the langauges in all the following segments . . . you need to understand their use!

[§2.5] Will help with understanding how the syntactical entities are represented.

[§3.1-3.2.6] Gives the foundation for the implementation techniques to be used later. [§3.2.7] onwards will be needed for Segment 3, so you can read it now, or leave it until then.

[Appendix B] Describes the scanning and parsing tools to be used — some of the detail is summarised from page 53 (Appendx B).

2.1 CORE

We shall start by creating a *very* simple programming language that consists of expressions, but has no way of naming values. This means that all values are interpreted *explicitly* — they will represent numbers and truth-values. We shall call this language CORE.[1]

The main goal of this segment is to create a basis for the more interesting languages which we shall see in later sections and segments, but concentrating on getting used to the tools and procedures that we shall be using (repetitively) as each new language or feature is developed.

2.1.1 Syntax

The syntax of CORE is:

Program	::=	*Expression*
Expression	::=	*Number*
	|	`-` `(` *Expression* `,` *Expression* `)`
	|	`zero?` `(` *Expression* `)`
	|	`if` *Expression* `then` *Expression* `else` *Expression*

Examples of programs that conform to this grammar are:

- `24`
- `-(3, 4)`
- `if zero?( 4 ) then 10 else 5`
- `if zero?(1) then 10 else if zero?(1) then 1 else 2`
- `if zero?(1) then 10 else -( 400, if zero?(1) then 1 else 2)`

From this grammar we can use the *EOPL* scanning and parsing tools to produce a syntax checker for this language. If you haven't done so already, read [Appendix B] which gives details of how to use SLLGEN to create parsers.

Exercise 17 Implement a parser for CORE. To do this:

1. *Copy* into your working directory the code the files
   - `syntax.scm`
   - `tests.scm`

   which are found in `/shared/rentedfs/cs-course/popl/Resources/Practicals/CurrentYear/CORE`. These (and other) files contain the basic structure for the implementation of CORE.[2]

[1] This does not appear explicitly in *EOPL*, but in fact it is the first language presented there, but with variables omitted. Segment 3 studies that language.

[2] For convenience, these files are listed in Appendix C. Note that they all begin with `#lang eopl`, the `scheme` dialect associated with *EOPL*, which you will be using from now onwards.

2. Open these in *DrRacket* ...I recommend using separate tabs, rather than separate windows.[3]

3. Replace the 'gaps' indicated by `???` with single or multiple expressions.

   Some points to note:

   - `(require file)` tells *DrRacket* to load the definitions exported by the string `file`
   - `(provide ...)` exports — makes visible — definitions from the file in which it is executed. `(provide (all-defined-out))` exports all definitions.
   - Some `(require ...)` expressions are commented out so that their code can be edited in subsequent exercises

   Don't forget to *save* your modifications as you proceed!

4. Test your parser by Runing `tests.scm`, and then executing `(scan&parse prog)` — defined in `syntax.scm` — where `prog` is one of the CORE programs (strings) defined in `tests.scm`.

   The output of `(scan&parse ...)` will be a display of the internal data-structure that represents the abstract syntax for the program. For example:

```
> (scan&parse Hmm?)
#(struct:a-program
  #(struct:zero?-exp #(struct:zero?-exp #(struct:const-exp 0))))
```

□

Note This is *not* a module about compilers (see SYAC), so only the simplest parsing tool (SLLGEN) is used. This will generate parsers for grammars which are called LL(1) — which is quite limiting — so the languages will be carefully designed to conform to this, and consequently might seem a little inflexible. To do better requires a more sophisticated parser generator, such as `yacc` (or its derivatives), which will construct parsers for the more general LALR(1) grammars.

Exercise 18 Create some more CORE programs in `tests.scm`, and test them.
□

2.1.2 Semantics

Producing a parser is interesting in itself, but it's not much use unless CORE programs can be *executed*. This implies that we need to process syntactically-correct programs — those that the parser has successfully analysed — to find their *value*, or *meaning*.

The *meaning* of a program is specified by means of a *semantic function*, which in these scripts will be called `(execute ...)`. This will call on another function which analyses the

[3]This behaviour can be set from the Edit|Preferences|General tab.

program in terms of its constituents, and constructs the final value. In these scripts, this will be called `(value-of ...)`.[4]

The exact nature of the `(value-of ...)` function will vary with the language being implemented, and this includes the number and type of the parameters. For the CORE language, the only argument will be a value which represents the *abstract syntax tree* of the program as supplied by the `(scan&parse ...)` function described in section 2.1.1

Expressed Values

Before being able to specify the `(value-of ...)` function, we need to be clear what we mean by a *value*. We need to distinguish between three classes of values: *Expressed*, *Denoted*, and *Storable*,[5] and so we need to provide *interfaces* to these kinds of data as explained in [Ch. 2]. We will make use of the `(data-structure ..)` and `(cases ...)` facilities provided in *DrRacket*, which you should review [§2.4] before continuing.

Exercise 19 Implement the *Expressed Values* data-type. To do this:

- *Copy* `data-structures.scm` from `/shared/rentedfs/cs-course/popl/Resources/Practicals/CurrentYear/CORE`.
- Open in a new tab in *DrRacket*.
- Replace the `???` with the correct code.

To *test* your solution, hit the Run button while in the tab that's displaying `data-structures.scm`. Since the code in this file doesn't `require` any other file's exports, it can be tested independently.

For some test examples, try evaluating (in the interactions window):

```
(->ExpVal 10)   (->ExpVal #t)  (->ExpVal "hello")  (<-ExpVal 10)
(<-ExpVal (->ExpVal 123))
```

. . . and others of your own.
□

Specifying `(value-of ...)`

Finally, you can implement the `value-of \ldots)` function. This should be done by *carefully* thinking how it is to provide the value of its argument in terms of the values of its argument's components. *Essentials of Programming Languages* describes how to approach this [§3.2.4–3.2.7], and you should follow their method. However, where the book refers to 'environments' that can be ignored for CORE . . . Segment 3 will incorporate this later.

Exercise 20 Complete the skeleton code so that the basic CORE language is implemented.

[4] You may, of course, call it anything you like. Typical alternatives might be `eval evaluate interpret interp meaning-of` etc. although at least one of these is already used by `scheme`, so redefining it might have strange effects!

[5] These will be/have been covered in the POPL lectures, and [§3.2.2], which you should review now.

- *Specify* the behaviour of `(value-of ...)` as shown in [§3.2.4–3.2.7] — this is a *paper* design task!
- *Copy* the remaining skeleton files from `/shared/rentedfs/cs-course/popl/Resources/Practicals/CurrentYear/CORE`, and open them in new tabs.
- Uncomment the remaining `(require ...)` calls from the files worked on in exercise 17.
- Replace the `???` with the correct code.

Hints

- Think *carefully* about what kind of value `(value-of ...)` should return.
- `(value-of ...)` can be used recursively.
- Keep the difference between *meta*-language, and *object*-language clear in your mind.

□

Testing You should test your solution against the example CORE programs in `test.scm`. This is most conveniently done using the `(run ..)` function defined in `driver.scm` which takes a program string, evaluates it, and returns the *scheme* value that is extracted from the *expressed* value produced by `(execute ...)` It's instructive to trace through the sequence of calls that occur.

2.2 Extending CORE

You now have a *complete* interpreter for basic CORE programs. All subsequent segments will use the same working method, and will be based on a similar set of files, and relationships between them. In fact, you will be able to take copies of a segment's solutions and then modify them to do the exercises in a later segment.

To demonstrate this, you should now extend CORE with some new facilities.

Exercise 21 *Extend* CORE to include the following numeric predicates which evaluate to *boolean* values: [*3.8*]

`equal?(`x`, `y`)`	is true iff $x = y$
`greater?(`x`, `y`)`	is true iff $x > y$
`less?(`x`, `y`)`	is true iff $x < y$

Test your extended version.

□

Exercise 21 shows that it can be tedious to add new primitive operations to a language, as they all follow a similar pattern. The following exercise asks you to *refactor*— change the implementation without altering the functionality — your CORE implementation in a particular way.

Exercise 22 *Refactor* the CORE code so that adding new primitive operators is easier. [*3.11*]

Test your modified version.

Hints

This can be done in a number of ways, but I've found that making use of some of the supplied `scheme` library functions for processing lists was very convenient. I recommend looking at the `(assoc ...)` function, and others in the so-called 'SRFI 1' library.

The description of these can be found by following the Help|Racket documentation item, searching for "srfi" in the browser window that opens, and then clicking through the 'SRFI 1' links.

To access these definitions you will need to add `(require srfi/1)` to the scheme code file that uses them.

□

[3.6, 3.7] **Exercise 23** *Extend* CORE as modified in Exercise 21 (or 22 if you chose to do it) to include the following operators which evaluate to *numeric* values:

`minus(`x`)`	evaluates to $-x$
`+(`x`,`y`)`	evaluates to $x + y$
`*(`x`, `y`)`	evaluates to $x \times y$
`/(`x`, `y`)`	evaluates to $x \div y$

Test it!

□

Segment 3

Names

In this segment you will build on the simple CORE language that you implemented in Segment 2. This will involve a single fundamental modification to CORE (introduction of names), and several other additions — both simple and challenging — to create a *new* more powerful language.

There will be *much* less 'housekeeping' detail given in this and subsequent segments: the workflow for progressing through the exercises will be the same as for Segment 2 — for each addition:

1. *Extend* or *modify* the *grammar* and *lexical specification* from an earlier version where necessary,
2. *Implement* a **parser** ... **test** and debug,
3. *Extend*, *implement*, or *refactor* the **interpreter function** ... **test** and debug.
4. Design and run even more tests!

Whenever the details given are incomplete or ambiguous, make suitable design decisions of your own. For the more challenging exercises some hints will be given, but there will normally be alternative ways of solving a problem, so don't be constrained by the hints.

As before, there will be *essential* exercises, which *must* be completed and some *optional* ones, which *should* be completed (or attempted!).

Reading

[§3.2] contains all you need for completing the fundamental exercises.

[§2.2] explains the implementation choices for *environments*.

3.1 LET

Start by taking copies of the files you used for CORE, and work with those. When testing your solutions, add new program examples to the `tests.scm` file, and Run from there — it'll save lots of typing later!

3.1.1 Syntax

The full syntax for LET is given in [Figs. 3.8, 3.9]. However, there is only one fundamental addition to CORE that LET introduces — the facility to *name values*. This involves adding two new productions to the syntax:

Expression **::=** ...
| *Identifier*
| `let` *Identifier* `=` *Expression* `in` *Expression*

This doesn't specify what an *Identifier* is. This *could* be done as part of the grammar, but it's more appropriate, and efficient, to make it part of the scanner.

Exercise 24

a) *Modify* the lexical specification used for CORE to recognise *identifiers.*[1]

You may use any sensible definition of what constitutes an identifier here, but try to make it as general as possible (but no more so). For instance, it could be as simple as a letter followed by a digit, or as general as that used for `scheme` identifiers.[2]

b) Test your lexer. This *can* be done without altering the parser specification!

□

[*Fig. 3.8 3.9*] **Exercise 25** *Implement* and test a *parser* for LET.
□

3.1.2 Semantics

Environments

To implement the semantics of LET we need to maintain an *environment* of *bindings* for the names so that their values can be retrieved when needed by the interpreter function. The book gives a detailed discussion of ways of implementing environments [§2.2, 3.2.3] which you should read.

For POPL purposes we can use *any* method without regard for efficiency, and so ...

Exercise 26 *Implement* and test the representation of *environments* as shown in [Fig. 2.1].

□

Exercise 27 *Implement* and *test* the interpreter functions for LET by *modify*ing the CORE
[§3.2.8] implementations of `(execute ...)` and `(value-of ...)`.

Hint

Be green — take care of your environment(s).
□

[1] Reread [Appendix B] if you're stuck!
[2] See R^nRS.

3.2 Extending LET

Exercise 28 *Extend* LET to incorporate a *multi-way conditional*. You should add the following production to the grammar: [*3.12*]

Expression ::= `cond` {*Expression* `==>` *Expression*}* `end`

The semantics should follow that of `scheme` (see R^nRS) except that, if *none* of the expressions on the left-hand sides of the `==>` selectors are true, then an error should be reported.[3]

Hint

Review the `arbno` or `separated-list` pattern keywords in the SLLGEN grammar specification [§B.3]. They can be awkward to use, and the corresponding case in `(value-of ...)` needs some careful thought. It's a bit like the `replace` / `many-replace` problem in section 1.3.1.
□

All serious languages provide ways of producing *composite* values, and currently we have no such facilities in LET. Rather than supplying many different ways of structuring data, we shall take the purist view, as exemplified by `scheme`, and provide a *single* structuring method: *lists*.

Exercise 29 *Extend* LET to incorporate *lists* of values as follows:[4]

a) Add expressions for creating and operating on lists to the language. These are defined as follows: [*3.9*]

`[]`	as `scheme '()`
`cons(`x`, `lst`)`	as `scheme (cons x lst)`
`car(`x`)`	as `scheme (car x)`
`cdr(`x`)`	as `scheme (cdr x)`
`null?(`x`)`	as `scheme (null? x)`

Hints

- Be *very* clear about the *meta* versus *object* language distinction.
- This can be tackled in several ways. I favour modifying the *Expressed Values* representation. Review how lists are defined in `scheme`— see R^nRS, or *SICP*— or in other languages such as Haskell.

b) Add a primitive expression which forms lists from multiple, comma-separated arguments. Since these lists will be another form of expression, syntax will be:[5] [≈*3.10*]

[3] How does this differ from the `scheme` definition?

[4] This exercise is challenging enough to be *optional*. However, if you don't add lists, only very dull programs can be written, especially when *recursive* definitions are introduced in Segment 4

[5] *EOPL* suggests a different syntax for this, but I prefer this way! See how easy it is to change the *look* of a language, without affecting its *meaning*! What does this tell you about the relative importance of syntax and semantics?

Expression ::= `[` *Expression* {, *Expression*}* `]`

Hint

- You might want to check the `(fold-right ...)`, and `(map ...)` functions from the srfi/1 list processing library mentioned in Exercise 22.

□

[≈3.15] **Exercise 30** *Extend* LET to have an operation `print` that takes one argument, *prints* it and returns the Expressed Value of its argument expression.
□

Currently the only way to specify multiple name bindings for an expression is to use *nested* `let` expression, such as:

```
let x = 10 in let y = -(10,3) in another = 42 in +( another, *(x, y))
```

which is annoying.

[3.16] **Exercise 31** *Modify* LET to allow `let` to accept any number of identifier bindings. The grammar should be extended with:

Expression ::= `let` {*Identifier* = *Expression*}* `in` *Expression*

The semantics should follow that of `scheme` (see R^nRS): each right-hand side is evaluated in the environment of the `let` expression. The body expression is then evaluated in the environment obtained by extending the environment of the `let` with the bindings specified.

Since the order in which the right-hand sides are evaluated is undefined, it should be an *error* for an identifier to appear on the left-hand side of more than one binding, however it is *not* an error for an identifier to appear in a right-hand side expression or expressions, and in (one) left-hand side.
□

Sometimes even the multiple-binding `let` is too restrictive, and a set of bindings needs to be made *in sequence*, as would be the case with nested `let`s.

[3.17] **Exercise 32** *Extend* LET with a *sequential* binding expression with the following syntax:

Expression ::= `let*` {*Identifier* = *Expression*}* `in` *Expression*

The semantics should, again, be that of `scheme`'s `let*` so:

- Each binding is executed in an environment extended by the 'earlier' bindings in the `let*`.
- It is *not* an error for an identifier to appear on more than one left-hand side.

Hint

- I found `fold-right` and `reverse`, from the srfi/1 library, useful.

□

Segment 4

Procedures

LET is a simple expression evaluator. For all practical programming though, it is severely limited — it's not even Turing-complete.[1] To be able to write any useful programs, we need the ability to *abstract* expressions — we need *procedures*, and we will want definitions of procedures to allow *recursion*.

This segment will *modify* LET in two stages: firstly by adding procedures, then by adding a binding construct for recursive definitions. Finally, a variation of the language in which the scoping rule differs from the 'standard' will be derived.

Since you will be developing *three* new languages from LET, there is a lot of work here. Much of this is explained in detail in *EOPL*, so you should allow time for reading the appropriate sections.

Reading

[§3.3] describes in depth how procedures are to be incorporated. Necessary for exercises 33 and 34.

[§3.4] details how recursive definitions are to be provided. Necessary for section 4.2.

[§3.5] is a good explanation of the 'standard' *lexical* scoping rule. This is an excellent back-up to the POPL lectures. Background for section 4.3.

4.1 PROC

4.1.1 Syntax

Procedures need to be *defined*, and *invoked*. So we need to add two new productions to the grammar to cover these two types of phrase. But the question arises: "What kind of syntactical phrase is a procedure definition?". Many old-fashioned languages put procedure definitions or declarations in a syntactic class called 'statements'. However, modern languages (and some ground-breaking older languages) classify procedure definitions as *expressions*, and we shall do the same.

The initial design will be for procedures to take *exactly one* argument (but see exercise 37), and for procedure invocation to follow the `scheme` form. Therefore the two new productions required for PROC are:

[1] Why not?

Expression ::= ...
| `proc (`*Identifier* `)`*Expression*
| `(`*Expression Expression* `)`

Exercise 33 *Derive* a new grammar for PROC from LET. Implement the scanner and parser, and test them against a variety of PROC programs.
□

4.1.2 Semantics

Introducing procedures has a number of significant effects on the language, and these are reflected in the semantics:

- Since procedures are the results of *expressions*, this type of value must be included in the langauges Expressed Values. This implies modifications to the `ExpVal` data-structure, which in turn requires decisions to be made on how procedure values are to be represented. Be clear about the *meta-* and *object*-language distinction (again).
- Procedure bodies must be evaluated in the correct environment which will, in general, be different from the environment in which the procedure was defined.

EOPL deals with these and other issues.

[§3.3] **Exercise 34**

a) *Derive* and implement an interpreter for PROC starting with that for LET.

b) Design a suite of test PROC programs, run them, and debug as necessary.

You should include in your tests (at least) the examples in [p.76, Ex 3.22, 3.25]

□

It's convenient to be able to define a procedure, and bind the value to a name in one statement. In fact, most langauges allow this, even if procedure value are not first-class.

[*3.19*] **Exercise 35** *Extend* PROC with a procedure binding construct, whose syntax is:

Expression ::= ...
| `letproc` *Identifier* `(`*Identifier* `)`*Expression* `in` *Expression*

The effect of a `letproc` expression such as:

```
letproc thing(x) .. in ...
```

should be the same as:

```
let thing = proc (x) .. in ...
```

□

Currying

It might seem that having single-parameter procedures is too limited for practical programming. However, as seen in the lectures,[2] procedures with any number of arguments can be programmed with single-parameter procedures, provided that procedure values are *first-class*.

For instance, if we could define procedures which take two arguments in PROC,[3] we might write:

```
letproc f(x y) ... in (f 10 23)
```

With single argument procedures, the same functionality can be obtained with:

```
letproc g(x) proc (y) ... in ((g 10) 23)
```

That is, `(g 10)` returns a procedure which, when applied to `23` give the same result as `(f 10 23)`. This technique is called *currying*. Since PROC procedures *are* first-class values, we can demonstrate this:

Exercise 36 Design and test a procedure, using Currying, which takes two arguments and multiplies them together. [*3.20*]

□

Even though Currying is all you need it is, perhaps, syntactically neater to allow procedure declarations with multiple parameters.

Exercise 37 *Modify* PROC so that procedure declarations can take multiple parameters: [*3.21*]

Expression	**::=**	...
	|	`proc (`{*Identifier*}* `)` *Expression*
	|	`(`*Expression* {*Expression*}* `)`

This is quite a challenging exercise!

Hints

- Be clear about what environment(s) to evaluate in.
- You choose whether or not to use separators (such as commas) in the parameter lists ... the above syntax shows *no* separators.
- Decide how to check that the number of argument in a procedure invocation is the same as in its declaration.

□

Most languages do not distinguish syntactically between calls to built-in procedures, such as `zero?(...)`, and user-defined procedures. This makes a program *look* neater! PROC differentiates between these.

Exercise 38 *Modify* the PROC language of exercise 34 or 35 so that the procedure call [*3.22*]

[2] ... and demonstrated in modern langauges like Haskell.

[3] We can't yet, but see exercise 37.

syntax is the same as that for primitives:

Expression **::=** ...
| *Identifier* `(` *Expression* `)`

This is much harder than would seem to be the case at first sight. If `SLLGEN` were more powerful than LL(1), then it would be relatively straight-forward. Interestingly, it is the *parser* that is hard to write here, and this complicates the design of the semantic function. Have a go!
□

When debugging programs, especially those where recursion is involved (see Section 4.2), it is often valuable to be able to see the sequence of calls to procedures, together with their arguments when called and their returned results.[4]

[3.27] **Exercise 39** *Extend* PROC to have a `traceproc` declaration which has the same effect as `proc`, except that when a procedure declared with `traceproc` is invoked, the value(s) of its argument(s) and its result are displayed as a 'side-effect'. It should behave exactly like a procedure declared with `proc` otherwise.

Test this by 'tracing' examples that you used as tests for PROC earlier: in particular, the example in [Ex 3.25] is 'interesting'.

Hint

- Review the `scheme (display ...)` procedure.
- Review the `scheme (begin ...)` special-form.

□

4.2 LETREC

The final piece in our language jigsaw[5] is to introduce self-referential definitions — otherwise known as *recursion*. *Syntactically* this seems trivial ... merely allow left-hand sides of `let` bindings to appear as components of the right-hand side expressions of the same binding: in fact, unless you've been really sophisticated with error checking, this is *syntactically* allowable in LET![6] However it's the semantics that get awkward. Specifically, it's deciding what *environments* the various expressions are evaluated in, and ensuring that they are produced correctly, that needs clear thinking.

4.2.1 Syntax

To ensure that the recursive bindings are easily distinguishable from the non-recursive ones — both to the programmer, and the interpreter — a new binding construct, `letrec`, is introduced with the following syntax:

[4] *DrRacket* `scheme`, and some other implementations, provide a `(trace ...)` special-form for doing this.
[5] Until we decide to change the picture!
[6] Try running `"letproc p(x) (p x) in 1"` — you even get the right answer! But now try `"letproc p(x) (p x) in (p 1)"` ...

Expression ::= ...
| `letrec` *Identifier* `=` *Expression* `in` *Expression*

4.2.2 Semantics

As suggested above, implementing the semantics of `letrec` involves concentrating on how to process the environments. This is dealt with in detail [§3.4],which you must read. Then the following exercise becomes trivial.

Exercise 40 *Derive* LETREC from PROC by adding the new production for `letrec` expressions and *modify*ing the interpreter.
□

You now have a powerful enough language to, finally, do the following:

Exercise 41 Implement the factorial function in LETREC. Enjoy!
□

Since Exercise 40 was so easy (given the explanation in *EOPL*), you should choose to do the following, optional, exercise.

Exercise 42 *Extend* LETREC so that the procedure declared in a `letrec` expression can have any number of arguments. [*3.31*]

The difficulties here are 'mere' `scheme` coding. By now you should be quite good at that!

Hints

- `fold-right` and `map` could be useful again.

□

The next stage is to have multiple bindings in a `letrec`

Exercise 43 [*3.32*]

a) *Extend* LETREC, or the extended version of Exercise 42, to allow the declaration of any number of mutually-recursive procedures.

b) Test the extended language on the program given in *EOPL* Exercise 3.32

□

Exercise 44 *Test* your language with the programs given in *EOPL* Exercises 3.23 and 3.25, confirming that they evaluate to the integer 12. [*3.23, 3.25*]
□

Exercise 45 Design, implement and test a LETREC program using the techniques of Exercise 44 which has the same behaviour as that of Exercise 43. [*3.24*]

If you decided not to attempt Exercise 43, perhaps you might like to re-consider?
□

[3.33] **Exercise 46** *Extend* the language of Exercise 43 to allow the declaration of *any number* of mutually recursive procedures of *any number* of arguments.
□

4.3 Scopes

You will by now be very familiar with the fact that the binding of a name can vary within a program. This means that whenever a name is to be evaluated, a programmer, or the interpreter implementation, must 'know' which binding to look at in order to decide the name's meaning. The *scope* of a binding determines where in the program the meaning of the name is that of the binding.

The vast majority of programming languages use a rule called *static*, or *lexical* scoping — this is the rule used by LETREC and its predecessors. In the final section of this segment you'll look at an alternative form of binding rules, called *dynamic scoping*.

Exercise 47 *Read* [§3.5] which expands on the material given in the POPL lectures.
□

The implementation of a particular scoping rule is a matter of deciding *which environment* contains the binding for a name when an expression using the name is evaluated. With *static* scoping, the meaning of a name is contained in the environment in force when the binding is encountered in the program *text*.[7] This includes the binding of names to *procedural* values — the meaning of the name is the meaning of the procedural definition, the *body* of which takes its meaning from the environment in force when the binding occurs.

However, static coping isn't the only choice. An alternative, sometimes (but rarely) seen, is to evaluate the procedure body in the environment in force *when invoked*, that is at the point when the procedure *is called*. This is called *dynamic binding*, or referred to as the *dynamic scoping* rule.

In many cases, the results obtained from static- and dynamic-binding are the same, but not always, as you will see in the following exercises.

[3.28] **Exercise 48**

1. Work out — by hand — what result this program would produce if LETREC were to use *dynamic scoping*:

```
let a = 3
in let p = proc(x) -(x,a)
       a = 5
   in -(a, (p 2))
```

[7] Hence the use of the term 'lexical' scoping.

2. Run this program with you r(statically-scopes) LETREC, and confirm that the results are different.

□

Exercise 49 [*3.28*]

a) Create a *new*[8] version of LETREC so that it uses *dynamic scoping* as described above.

b) Test the modified language by evaluating the program given in Exercise 48, and confirm that the result is what you predicted.

□

Dynamic binding *can* be useful, but introduces *significant* opportunities for obscure errors.

Exercise 50 Do [Ex 3.29] — work it out without running the example *first*! [*3.29*]
□

One interesting property of dynamic scoping is that it can be used to simulate recursive bindings — it becomes possible to use `let` 'recursively, without any need for `letrec`.

Exercise 51 Evaluate the following factorial program using both of your versions (static and dynamic) of LETREC: [*3.37*]

```
let fact = proc(n) +(n,1)
in let fact = proc(n)
                if zero?(n)
                   then 1
                   else *(n, (fact -(n,1)))
   in (fact 10)
```

If you did Exercise 39, try changing the `proc`s to `traceproc`s. Does this help you understand what's going on?
□

Exercise 52 Exercise [Ex 3.32] defines two mutually-recursive procedures which calculates whether a number is even or odd. Implement (and test) these *without* using `letrec`. [*3.37*]

□

4.4 Summary

You have implemented a usable, concise, complete functional programming language! It's not particularly 'efficient' in terms of time or storage use, but that is not the purpose of POPL ... SYAC will deal with some of those issues.[9]

[8] Keep the old statically-scoped version since you will be using it as the starting point for subsequent segments.

[9] Other parts of *EOPL* address these too.

If POPL were *only* about producing a complete programming language, we need go no further. However, there are various other principles of many programming langauges that need to be addressed, and these will be seen in the following segments.

Segment 5

State

THe langauges that have been implemented so far have the property that wherever a particular expression appears within a particular scope, it has the *same meaning* — this is the property of *Referential Transparency*. This boils down to the fact that a name either has *no* meaning due to it being *unbound*, or it has the meaning given by the binding in its 'closest' scope. Of course the meaning of a name *can* be altered by an inner `let` or `letrec`, but this alteration is only *local in scope*, with the 'earlier' meaning being restored (automatically) when the inner scope is exited.

In this segment you will see the effects of disposing with referential transparency in favour of being able to modify the meaning of names *globally*. This involves the introduction of *two* new concepts:

- References
- Sequencing

5.1 EXPLICIT-REFS

The modifications that will be required to LETREC to produce EXPLICIT-REFS are:

a) The extension of the set of *Expressed Values* to incorporate a new type of *References*

b) The implementation of an additional data-structure called a *store*, which provides a *globally modifiable memory*. This is used to model the idea of *mutable state*.

c) The introduction of three new primitives for dealing with references.

All these will involve modifications to the parser and semantic functions. *EOPL* is the prime source for understanding what is required, and therefore you should *read* [Ch.4]. The material in [§4.4] will not form part of the practicals, so that can be skipped if you like. [§4.5] will be covered in Segment 6.

Reading

[§4.1] motivates the consideration of *state* and its manipulation as an example of the wider concept of *computational effect*.

[§4.2.1] gives examples of how the semantics of LETREC expressions need to be re-specified for EXPLICIT-REFS.

5.1.1 Sequencing

For the previous langauges, the only concept of *sequence* of evaluation has been that an expression must be evaluated before its value can be used. This is implicit in the way the semantic function has been implemented. However, now that there is to be a store that can be mutated *globally*, the *order* in which some expressions are evaluated becomes significant, and so we need to be able to specify this order.

Syntax

EXPLICIT-REFS will introduce the very familiar *sequencing* construct:

Expression **::=** ...
| `begin` *Expression* {`;` *Expression*}* `end`

Exercise 53 *Modify* (your copy of) the LETREC parser to accept this `begin-end` construct, and test.
□

Semantics

THe `begin-end` sequencing construct is reasonably straight-forward but, to encourage careful thought:

[4.4] **Exercise 54** Specify, on paper, the behaviours of the `begin-end` expression.

Hints

- Use the methods for other specifications in *EOPL*,
- Consider the environment as always,
- Remember that it is an expression.

□

Now that you are clear about the sequencing construct's semantics ...

Exercise 55

a) *Implement* `begin-end` for EXPLICIT-REFS.

b) Test your implementation. There is only one language facility that can show this working!

□

5.1.2 References

Read [§4.2] if you've not already done so!

Syntax

The second syntactic addition will be the introduction of three new primitives:

Expression **::=** ...
| `&` (*Expression*) | `^` (*Expression*) | `:=` (*Expression* `,` *Expression*)

In *EOPL* these are presented as `newref`, `deref`, and `setref!` respectively. I prefer these shorter names.

Exercise 56 *Modify* the EXPLICIT-REFS parser from Exercise 53 to accept these primitives and test it.
□

Semantics

Before its possible to implement the semantics for these primitives, two things have to be done:

1. Design and implement (a representation of) the *store*, and
2. Extend the *expressed Values* data-type to incorporate a representatno of *references*

Both of these topics are well presented in *EOPL*, as you will have seen when you read [§4.2] earlier. So the following exercises should be relatively simple.

Exercise 57 *Modify* your *expressed Values* data-structure to include *references*. As will be seen in *EOPL*, the simple store model (Exercise **??**) expects integers as the representation of locations, so you need to be able to inject such numbers into `ExpVal` as the representation of references. [§4.2]

Test your code!
□

Exercise 58 Implement the simple *store* model given in *EOPL*. [*Fig. 4.1*] [*Fig. 4.2*]

Hints

- There may be some variations that will be needed depending upon how carefully you did Exercise 57.

□

Finally you can add clauses to the semantic function to evaluate the three new primitives. Read [§4.2.2] for their specification.

Exercise 59 [*Fig. 4.9*]

1. *Implement* the EXPLICIT-REFS primitives.

Hints

- [Fig. 4.9] gives you a start
- Don't forget that the store needs to be initialized (created).

2. Test with simple programs at first, but then *at least* those shown on [p.105].

□

Since *order of evaluation* is now a vital consideration, you should look at every place in the language definition where this was left unspecified before. In particular:

[*4.5, 4.11*] **Exercise 60**

1. Specify the `[ ... ]` list constructor, ensuring that the evaluation of its arguments or defined precisely.

2. Implement the specification.

3. Test it

What about procedure arguments?

□

Exercise 61

1. *Modify* EXPLICIT-REFS to use `{` and `}` instead of `begin` and `end`.

2. Test with all previous programs.

3. Trivial isn't it?

□

5.2 IMPLICIT-REFS

[§4.3]

Some, usually quite old, langauges follow the EXPLICIT-REFS model of providing primitives for explicitly dealing with references — the `C` family is probably the most commonly used. However most others use references *implicitly*, often in quite subtle ways — Java is an interesting modern example in which all *objects* are accessed via references.

In this section, the LETREC language will form the basis for IMPLICIT-REFS, in which *all* names[1] will be bound to references. These references will be created wherever there is a binding operation. Remember, binding occurs as a result of the `let`-type expressions *and* when procedure arguments are evaluated in procedure invocation.

[1] Such names are commonly-called *variables*, and this terminology will also be used here.

5.2.1 Syntax

There is only one *syntactical* addition to LETREC[2] to form IMPLICIT-REFS: the provision of an *assignment* primitive:

Expression **::=** ...
| `set` *Identifier* `=` *Expression*

Exercise 62 *Derive* a parser for IMPLICIT-REFS from LETREC by adding the production for `set`.
□

5.2.2 Semantics

EOPL gives a clear explanation of the sematnics for IMPLICIT-REFS, and guides you through its implementation.

Exercise 63 [§4.3]

a) *Implement* IMPLICIT-REFS

b) Test with [Ex.4.16]

□

If you have not done so already, you should add the extensions to IMPLICIT-REFS that you implemented for LETREC:

Exercise 64 *Extend* IMPLICIT-REFS to match your implementation of LETREC: [*4.17*] [*4.18*] [*4.19*]

- Multiple binding `let`
- Multi-argument procedures
- Multi-procedure `letrec` and/or `letproc`

□

5.3 Going Further

EOPL ends this journey with IMPLICIT-REFS by suggesting a change from an *expression-oriented* language to a *statement-oriented* language. This is the model followed by all *imperative* programming langauges — the sort that most people are familiar with.

Exercise 65 [*4.22*]

a) *Implement* the language defined in [Ex. 4.22], and test with the example programs given there.

[2]Remember: IMPLICIT-REFS is *derived* from LETREC, *not* EXPLICIT-REFS!

b) *Extend* this language according to the suggestions in [Ex 4.23–4.25]

□

Segment 6

Parameter Passing

Now that *references* have been introduced, it is possible to look in detail at the various argument-passing choices that language-designers have available, in addition to the common *call-by-value* that's been used up until now.

This segment will modify IMPLICIT-REFS in three ways to demonstrate:

- Call-by-*reference*
- Call-by-*name*, and
- Call-by-*need*

The surprising thing is, perhaps, how straight-forward it is to vary the basic language implementation to provide these new behaviours.

The description in *EOPL* is *very* clear, and takes you through this step-by-step, consequently this segment's text will be brief and will merely refer you to the book.

Reading

[§4.5] Read this from start to finish . . . it's only 8 pages!

6.1 BY-REFERENCE

Exercise 66 [§4.5.1]

1. Read [§4.5.1]
2. *Implement* BY-REFERENCE as detailed in [§4.5.1]
3. Test it.
4. Do [Ex 4.31–4.34]

□

Exercise 67 *Extend* BY-REFERENCE to incorporate *arrays* [*4.36*]
□

6.2 BY-NAME

Modern programming languages (and some pioneering older ones) are able to make good use of *Lazy Evaluation*. This comes in two varieties:

- Call-by-*name*
- Call-by-*need*

These are very well described in [§4.5.2], which demonstrates how easy it is to convert IMPLICIT-REFS into a lazy programming language.[1]

[§4.5.2]

Exercise 68

1. *Implement* BY-NAME as explained in *EOPL*.
2. *Implement* BY-NEED.

□

6.3 Going Further ...

There are many ways in which you could continue to work on the language(s) that you've developed Here are some suggestions

1. Try the *hard* exercises in *EOPL*. These are marked with [***] in the book.
2. Design a new, or several new, langauges that differ from the LETREC family in their *syntax*, but with *no change* to their semantics. Exercise 61 is one trivial example. You might find yourself constrained by the SLLGEN parser-generator technology of course, in which case make this a paper exercise. Alternatively, develop a better (LALR(k)) version of SLLGEN yourself![2]

   The important point to this is that it's the *semantics* not the *syntax* the defines a langauges power, or facilities etc., even though it's the syntax that makes it attractive (or not) at first sight.
3. Use you imagination!

[1] It is also pointed out that lazy evaluation is very awkward to use in the presence of mutable state — assignment — and so the resulting language should be used with care!

[2] That would be rated as [****], at least!

Appendix A

`scheme` Code Skeletons

Skeletons for the code for section 1.2 (you have to fill in the `???` with appropriate expressions):

```
(define (sentence)
  (append (append ??? (noun-phrase))    ;; ???
          (verb-phrase)
  )
)

(define (noun-phrase)
        ???                             ;; ???
)

(define (verb-phrase)
        ???                             ;; ???
)

(define (a-noun)
        ???                             ;; ???
)
(define (a-verb)
        ???                             ;; ???
)
(define (an-adjective)
        ???                             ;; ???
)
(define (an-adverb)
        ???                             ;; ???
)

(define (pick-random lst)
  (list-ref lst (random ??? ))          ;; ???
)

(define (either a b)
  (if  (= (random 2) 0)  (a) (b) )
)

(define noun-list (list 'dog 'cat 'student 'professor 'book 'computer))
```

```
(define verb-list (list 'ran 'ate 'slept 'drank 'exploded 'decomposed))

(define adjective-list (list 'red 'slow 'dead 'pungent 'over-paid 'drunk))

(define adverb-list (list 'quickly 'slowly 'wickedly 'majestically))
```

You may want to define other procedures ("helpers") in addition to these, for instance to help you structure your program more clearly, or to *abstract* common operations. Feel free to exercise your own judgement.

Appendix B

SLLGEN

This appendix is a summary of the SLLGEN parsing system taken from *Essentials of Programming Languages* Appendix B.[1]

In the following 'list' means a `scheme` list, and 'atom' means a `scheme` 'symbol' or atom (a 'quoted' name).

B.1 Scanning

A *scanner* or *lexer* specification is a *regular expression* represented as a list of atoms given by the following grammar:

scanner-spec	::=	`(` {*regexp-and-action*}* `)`
regexp-and-action	::=	*name* `(` {*regexp*}* `)` *action*
regexp	::=	*string* | `letter` | `digit` | `whitespace` | `any`
	|	`(not` *character* `)` | `(or` {*regexp*}*`)`
	|	`(arbno` *regexp* `)` | `(concat` {*regexp*}* `)`
action	::=	`skip` | `symbol` | `number` | `string`
name	::=	*atom*
string	::=	a `scheme` string
character	::=	a (unicode) character

The meaning and use of the terminal symbols (tokens) in this specification are explained in *EOPL* Appendix B.

B.2 Parsing

A *parser* specification is a BNF grammar represented as a list of atoms given by the following grammar:

grammar-spec	::=	`(` {*production*}* `)`
production	::=	`(` *lhs* `(`{rhs-item}* `)` *prod-name*
lhs	::=	*atom*
rhs-item	::=	*atom* | *string*
	|	`(arbno` {*rhs-item*}* `)`
	|	`(separated-list` {*rhs-item*}* *string* `)`
prod-name	::=	*atom*

[1] Which you must read.

The meaning and use of the `(arbno ...)` and `(separated-list ...))` forms in this specification are explained in *EOPL* Appendix B.

B.3 Generating scanners and parsers

There are several several procedures supplied that create scanners and parsers, and their associated data-structures — see *EOPL* for details. If we assume that `scanner-spec` and `grammar-spec` are lists of atoms that conform to the specifications in sections B.1 and B.2 respectively, then the two essential procedures are:

`(sllgen:make-string-parser scanner-spec grammar-spec)`

> Returns a procedure which takes a single string argument (the program). The returned procedure returns a data-structure representing the abstract syntax tree of the program, or an error if the program argument is syntactically incorrect.
>
> Errors are produced by `sllgen:make-string-parser` if there are syntax errors in either of the two specifications.

`(sllgen:make-define-datatypes scanner-spec grammar-spec)`

> This doesn't actually *return* anything as such,[2] rather it creates the `define-datatype` see definitions — section **??** — implied by your scanner and parser specifications. These definitions form the 'glue' which enables the scanner, parser and your own interpreter function — section 2.1.2 to work together. The various Segments show how how this all fits together.

Example

An example of using these three procedures is:

```
(define scanner '( ... ) )   ; where ... conforms to the scanner grammar
(define parser '( ... ) )    ; where ... conforms to the parser grammar
(sllgen:make-define-datatypes scanner parser) ;This is required!
(define scan&parse (sllgen:make-string-parser scanner parser))
```

Then, assuming you have implemented a `(execute ...)` function (section 2.1.2), evaluating a `scheme` expression such as

```
(execute (scan&parse "-( 23, pi);"))
```

will check that the string `-( 23, pi);` is syntactically correct, and if it is will return the meaning of the string interpreted according to your semantic specification.

[2] It is a *macro* which gets *expanded* ('evaluated') at *compile-time*. The `define-datatype` form is *also* a macro.

Appendix C

CORE skeletons

Skeleton files for CORE. This can also be found in:

`/shared/rentedfs/cs-course/popl/Resources/Practicals/CurrentYear/CORE`

`syntax.scm`

```
#lang eopl
(provide (all-defined-out))
;; Syntax for CORE
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; ** Requires editing the ??? ** ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Lexical structure
(define the-lexical-spec
  '(
    (whitespace (whitespace)                skip)
    (number     (???)                      number)    ;; ???
   )
)
; Grammar
(define the-grammar
  '(
    (program (expression) a-program)

    (expression (number)  const-exp)
    (expression
      ("zero?" "(" expression ")") zero?-exp)
    (expression
     ("-" "(" ??? ")")  diff-exp)     ;; ???
    (expression
     ("if" ??? "then" ??? "else" ???) if-exp)    ;; ???
    )
)
;;;;;;;;;;;;;;;; sllgen boilerplate ;;;;;;;;;;;;;;;;
;; Evaluating the following is *required* to construct the
;; data-types from the lexer and parser specs.
```

```
;; ... will be evaluated when this file is 'loaded'
(sllgen:make-define-datatypes the-lexical-spec the-grammar)

;; (scan&parse string) scans and parses the program represented
;; by the string argument. Produces an abstract syntax representation of
;; the program if no errors, which can be passed to (execute ...)
(define scan&parse
  (sllgen:make-string-parser the-lexical-spec the-grammar))
;; Applies the scanner given by lexical-spce to a string ...
;; used to test scanners
(define just-scan
  (sllgen:make-string-scanner the-lexical-spec the-grammar))
; (show-the-datatypes) displays the data-types created by the scanner
; and parser
(define (show-the-datatypes)
  (sllgen:list-define-datatypes the-lexical-spec the-grammar))
```

tests.scm

```
#lang eopl
(require "syntax.scm")
;(require "driver.scm")
;(require "semantics.scm")
;(require "data-structures.scm")
;;
;
; CORE
;
(define ten   "10")               ;; 10
(define true  "zero?(0)")         ;; #t
(define nope! "zero?(10)")        ;; #f
(define Hmm?  "zero?(zero?(0))")  ;; semantic error
(define HmHm!  "-( 2, zero?(2))") ;; semantic error
(define e1 ;; 3
    "if zero?( -( 2, 3) ) then 4 else -( 4, -(2,1))")
(define if1  ;; 2
     "if zero?(1) then 10 else if zero?(1) then 1 else 2")
(define if2 ;; 398
     "if zero?(1) then 10 else -( 400, if zero?(1) then 1 else 2)")
;
; ** Add more tests ***
```

driver.scm

```
#lang eopl
;; Loads all required pieces.
;; Provides the top-level (run ...) function
;;
```

```
(require "data-structures.scm")  ; for expval constructors
(require "syntax.scm")           ; for scan&parse
(require "semantics.scm")        ; for evaluate
;
(provide (all-defined-out))
;
;; run : String -> SchemeValue
(define (run string)
  (<-ExpVal
     (execute (scan&parse string))
  )
)
```

data-types.scm

```
#lang eopl
(provide (all-defined-out))
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; ** Requires editing the ??? ** ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;;
;; Expressed Values for CORE
;;    an expressed value is a number or a truth-value
;;
(define-datatype  ExpVal ExpVal?
    (number-ExpVal  (a-number number?))
    (bool-ExpVal    (a-boolean ???))    ;; ???
)
;; Injection function for taking a scheme value into the set of Expressed Values
(define (->ExpVal x)
  (cond
    ((number? x)  (number-ExpVal x))
    ((boolean? x) ??? )
  )
)
;;; Specific extractors:
; EOPL p70
; ExpVal->num : ExpVal -> number
(define (ExpVal->number v)
  (cases ExpVal v
    (number-ExpVal (s) s)
    (else (ExpVal-extractor-error 'Number v))
  )
)
; ExpVal->num : ExpVal -> truth-value
```

```
(define (ExpVal->bool v)
  (cases ExpVal v
    (bool-ExpVal (s) s)
    (else (ExpVal-extractor-error 'Boolean v))
  )
)
;; Convenience function for translating an Expressed value into a scheme value
(define (<-ExpVal x)
  (cases ExpVal x
    (number-ExpVal ??? ???)    ;; ???
    (bool-ExpVal   ??? ???)    ;; ???
  )
)
;;
(define (ExpVal-extractor-error variant value)
    (eopl:error 'ExpVal-extractors "Looking for a  s, found  s"
                variant value)
)
```

semantics.scm

```
#lang eopl
;; Semantic interpreter for CORE
(provide  (all-defined-out))
(require "syntax.scm")
(require "data-structures.scm")
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; ** Requires editing the ??? ** ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;;
;; (execute ...) takes an abstract syntax representation of a program,
;; and returns its Expressed Value
;;
(define (execute prog)
  (cases program prog
    (a-program (exp) (value-of exp))
  )
)
;;
;; (value-of ...) takes an abstract syntax representation of an expression
;; and returns its Expressed Value

;; ** Requires editing the ???s **
(define (value-of expr)
  (cases expression expr
```

```
    (const-exp (num) ??? )    ;; ???
    (diff-exp (exp1 exp2)
        (number-ExpVal (- ??? ???) )    ;; ???
    )
    (zero?-exp (exp)
       (??? (zero? ???))    ;; ???
    )
    (if-exp (test true-exp false-exp)
         (if (ExpVal->bool ???)    ;; ???
                   ???              ;; ???
                   ???            ;; ???
         )
    )
  )
)
```