1 Reading

- SICP, page 11 - 19, Section 1.1.6
- Little Schemer, Chapter 2

2 The Idea of Syntax Abstraction

Problem. Often programming tasks are repetitive, in that users (programmers) are doing same thing over and over again. Following are some examples of this repetitive tasks.

- Making local bindings,
- deeply nesting if expressions,
- iterating over a collection,
- accessing collection as an array, etc.

However, repetitive tasks are often tedious and error-prone. For example, local bindings can result in problems with inadvertant usage of unintended variables, deeply nesting if expressions can lead to logical errors, iterating over a collection often causes problems such as incorrect encoding of iterating over elements in the collection, explicitly programming access to a collection as an array often leaves inadvertant errors that result in array out of bound exceptions that only show up in corner cases, etc.

Solution. The common solution is to introduce new syntax that captures the pattern. The semantics for this new syntax can be given by translation to the original syntax.

Examples. Some examples of syntax abstraction in Scheme are let, letrec, and, or, cond, case. Some examples of syntax abstraction in C++/Java are for loops, foreach loops. Some examples of syntax abstraction in C# are delegate, indexers, property, etc.

Terms. There are four essential and related terms with respect to this idea: syntactic sugar, syntax abstraction, macro, and desugaring. Be sure to understand what each of these mean.

Exercise 2.1 How is this like procedural abstraction?

A syntactic abstraction also encapsulates a pattern of computation, but doesn’t need special syntax as a procedure works.

Exercise 2.2 Which is the sugar, the shorter, less tedious version, or the translation?

For the purposes of studying programming languages, it is important that we can “boil away” the sugars and be left with the essentials of a language, which is what we’ll focus on studying in this course.
3 Logical Connectives

**Problem.** The main issue with writing the code for emulate logical connectives is that it verbose and repetitive. For example, consider a function that evaluates to true when an element is a member of a list and false otherwise.

\[
\text{define member?} \\
\text{(lambda (e ls) } \\
\text{ (if (not (null? ls)) } \\
\text{ (if (equal? e (car ls)) } \\
\text{ #t } \\
\text{ (member? e (cdr ls)))) } \\
\text{ #f)})
\]

With logical connectives as syntactic abstraction in the language this could be written much more succinctly.

\[
\text{define member?} \\
\text{(lambda (e ls) } \\
\text{ (and (not (null? ls)) } \\
\text{ (or (equal? e (car ls)) } \\
\text{ (member? e (cdr ls))))})
\]

There are two terms with respect to logical connectives that deserve mention namely: short-circuit evaluation and conditional evaluation.

**Exercise 3.3** Can or evaluate all its arguments? What would happen?

No, it would loop. This shows the importance of short-circuit evaluation

3.1 Syntax and Semantics of Logical Connectives

The following grammar defines the syntax of logical connectives.

<expression> ::= (or {<expression>})* 
| (and {<expression>})* 
| ...

As mentioned before, semantics can be defined by translation to the original language (desugaring). Following listing shows an example. Note the use of variable arity procedures from the previous lectures.

\[
\text{(or ) = #f} \\
\text{(or e1 e2 ...) = (if e1 #t (or e2 ...))}
\]

\[
\text{(and ) = #t} \\
\text{(and e1 e2 ...) = (if e1 (and e2 ...) #f)}
\]

**Exercise 3.4** What’s this like in other languages? in C? in Pascal?

|| and && in C, not | and &. Pascal has no equivalent. cor, cand in Algol 68? and_then, or_else in Ada.

**Exercise 3.5** Rewrite the following using "and" and "or"
(define foo?
  (lambda (x)
    (if (pair? x)
        (if (number? (car x))
            #t
            (number? (cdr x)))
        #f))

Exercise 3.6  Define the predicate two-elem-list? to test whether a datum is a list of length two.

4  Cond Expression

Problem.  Usually nested ifs are hard to read. They also do not match the structure of input very well. For example, consider the following listing that count the number of variable references in an expression.

(define count-varrefs
  (lambda (exp)
    (if (varref? exp)
        1
        (if (lambda? exp)
            (count-varrefs
              (lambda->body exp))
            (+ (count-varrefs
                (app->rator exp))
               (count-varrefs
                (app->rand exp)))))

(define count-varrefs
  (lambda (exp)
    (cond ((varref? exp) 1)
          ((lambda? exp) (count-varrefs
                           (lambda->body exp)))
          (else (+ (count-varrefs
                     (app->rator exp))
              (count-varrefs
               (app->rand exp)))))))

Exercise 4.7  How would you write this in C or C++?

C++ doesn’t really provide something like cond, switch is not really the same, although similar but people make do with formatting conventions. Some languages, like Algol 68 and CLU, provide elseif as a keyword. An example of the usage of the elseif construct is provided in the following listing.

if b
  then s1()
elseif b2
  then s2()
else s3()
end
4.1 Syntax of Cond Expression

The following grammar shows the syntax of cond expression. Note the use of Kleene plus (+ / one or more) instead of Kleene star (*) / zero or more) in the production for expressions.

\[
<\text{expression}> ::= ( \text{cond} \newline
\hspace{1em} <\text{cond-clause}>^+ \newline
\hspace{1em} <\text{else-clause}> ) \newline
<\text{cond-clause}> ::= ( <\text{test}> <\text{body}> ) \newline
<\text{else-clause}> ::= ( \text{else} <\text{body}> ) \newline
<\text{test}> ::= <\text{expression}> \newline
<\text{body}> ::= <\text{expression}> \newline
\]

4.2 Translation-based Semantics of Cond Expression

\[
(\text{cond} \newline
\hspace{1em}(\text{test1 body1}) \newline
\hspace{1em}(\text{test2 body2}) \newline
\hspace{1em}... \newline
\hspace{1em}(\text{testn bodyn}) \newline
\hspace{1em}(\text{else bodye})) \newline
\]

Actually, a body can have any number of expressions, there’s an implicit begin there.

5 Case Expression

**Problem.** comparing the value of expression to many numbers, chars, symbols can be tedious. For example, consider the following listing that uses the cond expression.

\[
(\text{let} ((\text{pos (list-index x ls)})) \newline
\hspace{1em}(\text{cond} \newline
\hspace{2em}((\text{= pos 0}) 'first) \newline
\hspace{2em}((\text{= pos 1}) 'second) \newline
\hspace{2em}((\text{= pos 2}) 'third) \newline
\hspace{2em}((\text{or (= pos 3) (= pos 4)}) 'higher) \newline
\hspace{2em}(\text{else 'too-high}))) \newline
\]

An alternative implementation that uses case expression could be the following.

\[
(\text{case (list-index x ls)} \newline
\hspace{1em}((0) 'first) \newline
\hspace{1em}((1) 'second) \newline
\hspace{1em}((2) 'third) \newline
\hspace{1em}((3 4) 'higher) \newline
\hspace{1em}(\text{else 'too-high}))) \newline
\]

**Exercise 5.8** What’s this like in C/C++? In Pascal?

This is like switch, Pascal’s case

**Exercise 5.9** What’s different about this from the C/C++ statement?
It is an expression, therefore no “break” is needed. This helps avoid a common error in the use of switch statements.

**Historic Perspective.** The case expression was invented by Tony Hoare. Pascal was the first language to have this feature. Usually this has very fast implementation, typically uses indirect branch through jump table based on value.

### 5.1 Syntax of Case Expression

The following shows the grammar for case expressions.

$$
<expression> ::= ( case <expression> \\
                   {<case>}% \\
                   <else-clause> ) \\
<case> ::= ( <key-list> <body> ) \\
<key-list> ::= ( {<key>}+ ) \\
<key> ::= <symbol> | <number> | <char> \\
<else-clause> ::= ( else <body> ) \\
<body> ::= <expression>
$$

### 5.2 Translation-based Semantics of Case Expressions

```scheme
(let ((*key* e))
  (case e (*cond 
             (k11 b1) ((memv *key* 'k11) b1) 
             (k12 b2) ((memv *key* 'k12) b2) 
             ... 
             (kln bn) ((memv *key* 'k13) b3) 
             (else be))) 
     (else be)))
```

### 6 Local Bindings

For this section, you are encouraged to see R5RS for details. The required reading for this section is SICP pp. 63–64.

### 6.1 The Problems Solved by Let

```scheme
(define double-reverse-sublist 
  (lambda (lst) 
    (if (null? lst) 
        () 
        (cons (reverse (car lst)) 
              (cons (reverse (car lst)) 
                    (double-reverse-sublist 
                                 (cdr lst)))))))
```

The main problems with this code are the following.
- loss of time by repeating computations
- less clear, tedious to repeat code
- with side effects, need to save value
Exercise 6.10  What’s the time complexity of reverse?

Exercise 6.11  How much slower is the code than it would be if didn’t have to repeat that computation?

Exercise 6.12  What mechanism do we have, so far, that binds names to values?

6.2  Rewrites to Avoid Repeated Computation

(define double-reverse-sublist
  (lambda (lst)
    (if (null? lst) '()
        (cons-twice
          (reverse (car lst))
          (double-reverse-sublist (cdr lst))))))

(define cons-twice
  (lambda (elem ls)
    (cons elem (cons elem ls))))

But it’s annoying to have to use a helping procedure. We could eliminate it, since it’s only used once, by substituting its definition.

6.3  Substituting the Definition of cons-twice

(define double-reverse-sublist
  (lambda (lst)
    (if (null? lst) '()
        ((lambda (elem ls)
            (cons elem (cons elem ls)))
          (reverse (car lst))
          (double-reverse-sublist (cdr lst))))))

But this is hard to read. Why?

Exercise 6.13  What do other languages do to avoid recomputation?

Variables, assignment, constant declarations. Primary reason is that it is better for humans to have the names, like elem and ls, bound to values first, close to each other.

6.4  Using Let

In Scheme similar effect can be achieved by using the let expression. An example rewrite of the double reverse sublist function using let follows.
(define double-reverse-sublist
  (lambda (lst)
    (if (null? lst) '()
      (let {
          (elem (reverse (car lst)))
          (ls (double-reverse-sublist (cdr lst)))
        }
        (cons elem (cons elem ls)))
      )))

An alternative implementation is as follows.

(define double-reverse-sublist
  (lambda (lst)
    (if (null? lst) '()
      (let {
          (elem (reverse (car lst)))
          (cons elem (double-reverse-sublist (cdr lst)))))
    )))

So this is the idea, we define the meaning of let by translating it back to that hard-to-read lambda application, humans get nice syntax, language gets simple semantics.

Exercise 6.14  What is this like in other languages?

Declaration forms (in C, Pascal), so this is a kind of local declaration construct.

6.5 Syntax of Let

<Scheme-exp> ::= |
  (let ({<binding>}+) <body>)
<binding> ::= (<var> <Scheme-exp>)
<body> ::= <Scheme-exp>

The production for <body> is a bit of a lie, it’s actually a list of <Scheme-exp>, but we won’t see why until we discuss side effects.

6.6 Semantics of Let

(let ((var1 expl1)
      ...
      (varn expn))
  body)

= 

((lambda (var1 ... varn)
  body)
  expl1 ... expn)

This is where the two parentheses come from. To illustrate the process, let us look at an example desugar of the following function using let expression.

(define addem
  (lambda (x y z)
    (let {{x (+ y z))
           (y (+ x 0))
           (+ x y)!!})
      )
      )
      )
which desugars into the following function without let expression.

\[
\text{(define addem2}
\text{(lambda (x y z)
  ((lambda (x y)
    (+ x y))
  (+ y z)
  (+ x 0)))))
\]

Exercise 6.15  What is this equal to?

Exercise 6.16  [For You To Do (IN PAIRS)] What is the difference between the following?

\[
\text{(define x 9)}
\text{(let ((x 3))}
  \text{(let ((y (+ x 4)))}
    (* x y)))
\]

and

\[
\text{(define x 9)}
\text{(let ((x 3)}
  \text{(y (+ x 4)))}
  (* x y))
\]

Hints: It may help to convert each of these into their desugared form? It also helps to draw arrows from variable references to the variable declarations.

Answer: In the first one, the x’s refer to 3, while in the second let, the x refers to 9 so the first is 21, the second is 33.

Exercise 6.17  How would you define the region of a var decl in a let? the scope?

7  The LETREC Expression

The reading for this section is SICP p. 391 and EOPL p. 23.

7.1  What does let helps us with?

It helps us avoid namespace clutter. Names are defined closer to where they are used and their scope is limited.

\[
\text{(define circle-area}
\text{(let ((pi 3.14159))}
  \text{(lambda (radius)}
    (* pi (* radius radius)))))
\]

So you might like to do that for helping procedures. Why? It would certainly help with managing procedure names in the program, if we don’t need the helping procedures else where in the program.
7.2 Does Let Work for Procedures?

\begin{verbatim}
(define list-product
 (let
  ((prod-iter
    (lambda (lon acc)
      (if (null? lon) acc
        (if (zero? (car lon)) 0
          (prod-iter (cdr lon) (* acc (car lon))))))))
  (lambda (lon)
    (prod-iter lon 1))))
\end{verbatim}

**Exercise 7.18** What does the recursive call of prod-iter refer to?

Nothing!

**Exercise 7.19** Why doesn’t that work?

The scope of prod-iter doesn’t include its body!

7.3 We Need to Introduce a New Expression (LETREC)

\begin{verbatim}
(define list-product
 (letrec
  ((prod-iter
    (lambda (lon acc)
      (if (null? lon) acc
        (if (zero? (car lon)) 0
          (prod-iter (cdr lon) (* acc (car lon))))))))
  (lambda (lon)
    (prod-iter lon 1))))
\end{verbatim}

7.4 Syntax

\begin{verbatim}
<Scheme-exp> ::= 
  | (letrec {{<pbinding>}+}
      <body>)
<pbinding> ::= (<var> <lambda-exp>)

<lambda-exp> ::= (lambda <formals> <body>)
<formals> ::= {{<var>}+} | <var>
<body> ::= <Scheme-exp>
\end{verbatim}

**Exercise 7.20** How are local declarations of recursive procedures handled in other languages? For example, Pascal, ML.

**Exercise 7.21** Is there any difference from Scheme?

They aren’t allowed within expressions in other languages.
Exercise 7.22  [FOR YOU TO DO (IN PAIRS)] Using letrec define a combination of mutually recursive functions even? and odd? with the expected semantics.

(letrec ((odd? (lambda (n) (and (not (zero? n)) (even? (- n 1)))))
      (even? (lambda (n) (or (zero? n) (odd? (- n 1))))))
  (list (odd? 22?) (even? 342)))
=> (#t #t)
%

7.5  Advantages and Disadvantages of Using LETREC

The main disadvantage of using letrec expression is that it may be harder to debug the internal procedures. The advantages are as follows.

- namespace control
- avoid passing
- often optimal (as it can avoid passing arguments)

Exercise 7.23  [DO-IN-PAIRS] Using a letrec, define the function expt, which takes two numbers x and y and computes $x^y$.

{expt 4 0} = 1
{expt 3 1} = 3
{expt 3 2} = 9
{expt 3 3} = 27

Exercise 7.24  What is the region of a variable definition in a let compared to a letrec?

In a let the region of the variable declarations v1 = E1, v2 = E2, ... , vn = En includes only the body expression. Whereas, in a letrec the region of the variable declarations v1, v2, ... , vn includes the expressions E1, E2, ... , En, and the body expression.

Exercise 7.25  What are the advantages of using letrec?
There are several advantages. First, it is sometimes faster (as it can avoid passing arguments repeatedly). Second, it allows control of global namespace, and third, it allows access control (no one outside can use local procedure)

Exercise 7.26 Any disadvantages?

It is a bit difficult to test the local helping procedures!