Contracts and types

Our contracts describe the type of data consumed by and produced by a function.

Until now, the type of data was either a basic (built-in) type, a defined (struct) type, an anyof type, or a list type, such as List-of-Symbols, which we then called (listof Sym).

Now we need to talk about the type of a function consumed or produced by a function.
Key Point: We can use the contract for a function as its type.

For example, the type of $>$ is (Num Num → Bool), because that’s the contract of that function.

We can then use type descriptions of this sort in contracts for functions which consume or produce other functions.
An example:

```
(define trans-table (list (list '+ ' +)
                      (list '∗ ' ∗)))
```

;; (lookup-al k alst) finds the value in alst corresponding to key k
;; lookup-al: Sym (listof (list Sym (Num Num → Num))) →
;;       (anyof false (Num Num → Num))

(define (lookup-al k alst)
  (cond [(empty? alst) false]
        [(equal? k (first (first alst))) (second (first alst))]
        [else (lookup-al k (rest alst))])))
Contracts for abstract list functions

filter consumes a function and a list, and produces a list.

We might be tempted to conclude that its contract is

\[(\text{Any} \rightarrow \text{Bool}) \ (\text{listof Any}) \rightarrow (\text{listof Any}).\]

But this is not specific enough.

Consider the application \((\text{filter odd? (list 1 2 3)})\). This does not obey the contract (the contract for odd? is \(\text{Int} \rightarrow \text{Bool}\)) but still works as desired.

*The problem*: there is *a relationship between the two arguments* to filter and the result of filter that we need to capture in the contract.
Parametric types

An application \( \text{filter pred? lst} \), can work on any type of list, but the predicate provided should consume elements of that type of list.

*Key Point:* In other words, we have *a dependency* between the *type of the predicate* (which is the contract of the predicate) and the *type of list*.

To express this dependency, we *use a type variable*, such as \( X \), and use it in different places to indicate where the same type is needed.
The contract for filter

filter consumes a predicate with contract \((X \rightarrow \text{Bool})\), where \(X\) is the base type of the list that it also consumes.

It produces a list of the same type it consumes.

The contract for filter is thus:

\[
;; \text{filter: (X} \rightarrow \text{Bool) (listof X) } \rightarrow \text{ (listof X)}
\]

Here \(X\) stands for the unknown data type of the list.

We say filter is polymorphic or generic; it works on many different types of data.
The contract for `filter` has three occurrences of a type variable `X`. Here the type variable is used to indicate a relationship. In other cases (e.g. Module 6) when we talk about a list, we might use the term `(listof X)` to mean the elements of the list are all the same type. We will soon see examples where more than one type variable is needed in a contract.
Using contracts to understand

*Key Point:* Many of the difficulties one encounters in using abstract list functions can be overcome by careful attention to contracts.

For example, the contract for the function provided as an argument to `filter` says that it consumes one argument and produces a Boolean value.

This means we must take care to never use `filter` with an argument that is a function that consumes two variables, or that produces a number.
Simulating structures *Not covered this term*

We can use the ideas of producing and binding functions to simulate structures.

```
(define (my-make-posn x y)
  (local
    [(define (symbol-to-value s)
        (cond [(symbol=? s 'x) x]
              [(symbol=? s 'y) y]))
     symbol-to-value))

A trace demonstrates how this function works.
```
Simulating structures *Not covered this term!*

\[
\text{(define p1 (my-make-posn 3 4)) ⇒}
\]

\[
\text{(define p1 (local}
\]
\[
\text{[(define (symbol-to-value s)
\]
\[
\text{(cond [(symbol=\? s 'x) 3]
\]
\[
\text{[(symbol=\? s 'y) 4]])]
\]
\]
\text{symbol-to-value))}
\]

Notice how the parameters have been substituted into the local definition.

We now rename `symbol-to-value` and lift it out.
Simulating structures *Not covered this term!*

This yields:

```
(define (symbol-to-value_38 s)
  (cond [(symbol=? s 'x) 3]
        [(symbol=? s 'y) 4]))
```

(define p1 symbol-to-value_38)

`p1` is now a function with the `x` and `y` values we supplied to `my-make-posn` coded in.

To get out the `x` value, we can use `(p1 'x)`:  

```
(p1 'x) ⇒ 3
```
Simulating structures *Not covered this term*

We can define a few convenience functions to simulate `posn-x` and `posn-y`:

```
(define (my-posn-x p) (p 'x))
(define (my-posn-y p) (p 'y))
```

If we apply `my-make-posn` again with different values, it will produce a different rewritten and lifted version of `symbol-to-value`, say `symbol-to-value_39`.

We have just seen how to implement structures without using lists.
Simulating structures *Not covered this term*

Our trace made it clear that the result of a particular application, say `(my-make-posn 3 4)`, is a “copy” of `symbol-to-value` with 3 and 4 substituted for `x` and `y`, respectively.

That “copy” can be used much later, to retrieve the value of `x` or `y` that was supplied to `my-make-posn`.

This is possible because the “copy” of `symbol-to-value`, even though it was defined in a `local` definition, survives after the evaluation of the `local` is finished.
Anonymous functions

(define (make-adder n)
  (local [(define (f m) (+ n m))]
    f))

The result of evaluating this expression is a function.

What is its name? It is **anonymous** (has no name).

When you do calculations such as \((\ast (\div 2 3) 4)\) the intermediate result \(5\) also does not have a name.

This is sufficiently valuable that there is a special mechanism for it.
Producing anonymous functions

(define (not-symbol-apple? item) (not (symbol=? item 'apple)))
(define (eat-apples lst) (filter not-symbol-apple? lst))

This is a little unsatisfying, because not-symbol-apple? is such a small and relatively useless function.

It is unlikely to be needed elsewhere.

We can avoid cluttering the top level with such definitions by putting them in local expressions.
Using local

(define (eat-apples lst)
  (local [(define (not-symbol-apple? item)
            (not (symbol= item 'apple)))]
    (filter not-symbol-apple? lst)))

This is as far as we would go based on our experience with local.

But now that we can use functions as values, the value produced by the local expression can be the function not-symbol-apple?.

We can then take that value and deliver it as an argument to filter.
(define (eat-apples lst)
  (filter (local [(define (not-symbol-apple? item)
                   (not (symbol =? item 'apple)))]
           not-symbol-apple?)
     lst))

But this is still unsatisfying. *Why should we have to name not-symbol-apple? at all? In the expression \((\times (\mathbf{+} 2 3) 4)\), we didn’t have to name the intermediate value 5.*

Racket provides a mechanism for *constructing a nameless function* which can then be used as an argument.
Introducing lambda

(local [(define (name-used-once x1 ... xn) exp)]
  name-used-once)

can also be written

(lambda (x1 ... xn) exp)

lambda is used to create anonymous functions.

lambda can be thought of as “make-function”.

Key Point: It can be used to create a function which we can then use as a value – for example, as the value of the first argument of filter.
We can then replace

```
(define (eat-apples lst)
  (filter (local [(define (not-symbol-apple? item)
                  (not (symbol=? item 'apple)))]
              not-symbol-apple?)
         lst)
```

with the following:

```
(define (eat-apples lst)
  (filter (lambda (item) (not (symbol=? item 'apple))) lst))
```
**lambda** is available in Intermediate Student with Lambda, and discussed in section 24 of the textbook.

We’re jumping ahead to it because of its central importance in Racket, Lisp, and the history of computation in general.

The designers of the teaching languages could have renamed it as they did with other constructs, but chose not to out of respect.

The word **lambda** comes from the Greek letter, used as notation in the first formal model of computation.
We can use \texttt{lambda} to simplify \texttt{make-adder}. Instead of

\begin{verbatim}
(define (make-adder n)
  (local [(define (f m) (+ n m))]
    f))
\end{verbatim}

we can write:

\begin{verbatim}
(define (make-adder n)
  (lambda (m) (+ n m)))
\end{verbatim}

\textit{Key Point:} \texttt{lambda} replaces \texttt{local}, \texttt{define} and the identifier \texttt{foo} of an anonymous function but keeps the argument(s) and body.
lambda also underlies the definition of functions.

Until now, we have had two different types of definitions.

;; a definition of a numerical constant
(define interest-rate 3/100)

;; a definition of a function to compute interest
(define (interest-earned amount)
  (* interest-rate amount))

There is really only one kind of define, which binds a name to a value, where the value may be a function.
Internally, 

\[
\text{(define (interest-earned amount)}
\text{ (\ast interest-rate amount))}
\]

is translated to

\[
\text{(define interest-earned}
\text{ (lambda (amount) (\ast interest-rate amount)))}
\]

which binds the name \text{interest-earned} to the value \text{(lambda (amount) (\ast interest-rate amount))}. 
We should change our semantics for function definition to represent this rewriting.

But doing so would make traces much harder to understand.

As long as the value of defined constants (now including functions) cannot be changed, we can leave their names unsubstituted in our traces for clarity.

In stepper questions, if a function is defined using function syntax, you can skip the lambda substitution step. If a function is defined as a constant using lambda, you must include the lambda step.
For example, here’s make-adder rewritten using lambda.

```
(define make-adder
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

What is ((make-adder 3) 4)?
(define make-adder (lambda (x) (lambda (y) (+ x y))))

((make-adder 3) 4) ⇒ ;; substitute the lambda expression

(((lambda (x) (lambda (y) (+ x y))) 3) 4) ⇒

((lambda (y) (+ 3 y)) 4) ⇒

(+ 3 4) ⇒ 7

make-adder is defined as a constant using lambda, so it is
substituted in place of make-adder.
Syntax and semantics of lambda

Before

*First position* in an application must be a built-in or user-defined function

A function name had to follow an *open parenthesis*.

Now

*First position* can be an expression (computing the function to be applied). Evaluate it along with the other arguments.

A function application can have *two or more open parentheses* in a row: `((make-adder 3) 4)`.
**Semantics of lambda**

*Key Point:* We need a rule for evaluating applications where the function being applied is anonymous (a lambda expression.) The rule for evaluating

\[(\text{lambda} \ (x_1 \ldots \ x_n) \ \text{exp}) \ v_1 \ldots \ v_n \Rightarrow \text{exp'}\]

is that \(\text{exp'}\) is \text{exp} with all occurrences of \(x_1\) replaced by \(v_1\), all occurrences of \(x_2\) replaced by \(v_2\), and so on.

As an example:

\[(\text{lambda} \ (x \ y) \ (\ast \ (\mathop{+} \ y \ 4) \ x)) \ 5 \ 6 \Rightarrow \ (\ast \ (\mathop{+} \ 6 \ 4) \ 5)\]
Suppose during a computation, we want to specify some action to be performed one or more times in the future.

*Before* knowing about lambda, we might build a *data structure* to hold a description of that action, and a *helper function* to consume that data structure and perform the action.

*Now*, we can just describe the computation clearly using lambda.
Example: character translation in strings

We’d like a function, `translate`, that translates one string into another according to a set of rules that are specified when it is applied.

In one application, we might want to change every instance of ’a’ to a ’b’. In another, we might translate lowercase characters to the equivalent uppercase character and digits to ’*’.

(\texttt{check-expect (translate "abracadabra" \ldots) "bbrbcdbbbrb"})
(\texttt{check-expect (translate "Testing 1-2-3" \ldots) "TESTING *-*-*"})

We use \ldots to indicate that we still need to supply some arguments.
We could imagine `translate` containing a `cond`:

```
(cond [(char = ? ch #\a) #\b]
     [(char-lower-case? ch) (char-upcase ch)]
     [(char-numeric? ch) #\*]
     ...
)
```

But this fails for a number of reasons:

- The rules are “hard-coded”; we want to supply them when `translate` is applied.
- A lower case ‘a’ would always be translated to ‘b’; never to ‘B’

But the idea is inspiring...
**Goal:** develop a *general method* of performing character translations on strings.

Suppose we supplied `translate` with a list of question/answer pairs:

```
;; A TranslateSpec is one of:
;; * empty
;; * (cons (list Question Answer) TranslateSpec)
```

Like `cond`, we could work our way through the `TranslateSpec` with each character. If the Question produces `true`, then apply the Answer to the character. If the Question produces `false`, go on to the next Question/Answer pair.

What are the types for `Question` and `Answer`?
Functions as first class values can help us. Both \texttt{Question} and \texttt{Answer} are functions that consume a \texttt{Char}.

\texttt{Question} produces a \texttt{Bool} and \texttt{Answer} produces a character. This completes our data definition, above:

\begin{verbatim}
;; A Question is a Char \rightarrow Bool
;; An Answer is a Char \rightarrow Char
\end{verbatim}

And a completed example:

\begin{verbatim}
(check-expect (translate "Testing 1-2-3"
               (list (list char-lower-case? char-upcase)
                     (list char-numeric? (lambda (ch) #\*))))
"TESTING *-*-*")
\end{verbatim}
Translate: developing the code

translate consumes a string and produces a string but we need to
operate on characters. This suggests a wrapper function:

;; A TranslateSpec is one of:
;; * empty
;; * (cons (list Question Answer) TranslateSpec)

;; (translate s spec) translates s according to the spec
;; translate: Str TranslateSpec → Str

(define (translate s spec)
  (list→string (trans-loc (string→list s) spec)))
(check-expect (trans-loc (list #\a #\9))
    (list (list char-lower-case? char-upcase))) (list #\A #\9))

(define (trans-loc loc spec)
  (cond [(empty? loc) empty]
        [(cons? loc) (cons (trans-char (first loc) spec)
                             (trans-loc (rest loc) spec))])))

(define (trans-char ch spec)
  (cond [(empty? spec) ch]
        [((first (first spec)) ch) ((second (first spec)) ch)]
        [else (trans-char ch (rest spec))])))
(check-expect (translate "Testing 1-2-3"
  (list (list char-lower-case? char-upcase)
      (list char-numeric? (lambda (ch) #\*)�)
  )
) "TESTING *-*-*-")

(check-expect (translate "abracadabra"
  (list (list (lambda (ch) (char= ch #\a))
          (lambda (ch) #\b)))
) "bbrbcdbbrb"

The repeated lambda expressions suggest some utility functions:

(define (is-char? c1) (lambda (c2) (char=? c1 c2)))
(define (always c1) (lambda (c2) c1))
Abstracting another set of examples

Here are two early list functions we wrote.

(define (negate-list lst)
  (cond [(empty? lst) empty]
        [else (cons (− (first lst)) (negate-list (rest lst)))]))

(define (compute-taxes payroll)
  (cond [(empty? payroll) empty]
        [else (cons (sr→ tr (first payroll))
                    (compute-taxes (rest payroll)))]))
We look for a difference that can’t be explained by renaming (it being what is applied to the first item of a list) and make that a parameter.

I.e. in both cases we are applying a function \( f \) to each element of the list, so consume \( f \) as a parameter.

;; (may-map f lst) applies f to every element of lst
(define (my-map f lst)
  (cond [(empty? lst) empty]
        [else (cons (f (first lst))
                     (my-map f (rest lst))))]))
Tracing my-map

(my-map sqr (list 3 6 5))
⇒ (cons 9 (my-map sqr (list 6 5)))
⇒ (cons 9 (cons 36 (my-map sqr (list 5))))
⇒ (cons 9 (cons 36 (cons 25 (my-map sqr empty))))
⇒ (cons 9 (cons 36 (cons 25 empty)))

my-map performs the general operation of transforming a list element-by-element into another list of the same length.
Using my-map

The application

\[(\text{my-map } f \ (\text{list } x_1 \ x_2 \ldots \ x_n))\]

has the same effect as evaluating

\[(\text{list } (f \ x_1) \ (f \ x_2) \ldots \ (f \ x_n)).\]

We can use my-map to give short definitions of a number of functions we have written to consume lists:

\[
(\text{define } (\text{negate-list lst}) \ (\text{my-map } - \ \text{lst}))
\]

\[
(\text{define } (\text{compute-taxes lst}) \ (\text{my-map sr \rightarrow tr \ lst}))
\]

How can we use my-map to rewrite trans-loc?
The contract for my-map

my-map consumes a function and a list, and produces a list.

How can we be more precise about its contract, using parametric type variables?