CSC324: Principles of Programming Languages

Lecture 8

Wednesday, 3 June, 2020

Comments on lab 4

- People seemed to like the recitation format! We'll stick with it going forward.
- If you didn't find the earlier labs a good use of your time, you might like the new format better.

Comments on exercise 3

Many of you only implemented functionality that destructures only one kind of Expr

```haskell
data Expr
  = Number Integer         -- ^ A numeric literal.
  | Identifier String      -- ^ An identifier.
  deriving (Show, Eq)

data Binding
  = Binding String Expr -- ^ A binding of an identifier name to an expression.
  deriving (Show, Eq)

someFunc Binding s (Number n) = ...
-- What about if the expression isn't a number?
```

Comments on exercise 3

Many of you only implemented functionality that destructures only one kind of Expr

```
data Expr
  = Number Integer        -- ^ A numeric literal.
  | Identifier String     -- ^ An identifier.
  deriving (Show, Eq)

data Binding
  = Binding String Expr -- ^ A binding of an identifier name to an expression.
  deriving (Show, Eq)

someFunc Binding s (Number n) = ...
someFunc Binding s (Identifier id) = ...
```

In previous classes...

… we discussed how **quotation** allows a strictly-evaluated language to change an expression's evaluation semantics, and how we call an expression with non-standard evaluation rules a **special form**.

Today, we'll talk about how to implement special forms of our own

Zzzzzz…..

How long will it take for this call to snooze to return?

Who knows??? We don't have units, argh

```
#lang racket

; sleep pauses the execution of the program
; for the amount of time given.
;
; int -> void
(define (sleep n)
  ...)

; it's 12:00:00 right now.  When do we resume?
(sleep 60)
```

Zzzzzz…..

One way to solve this is to make it explicit in the documentation how long we'll sleep for

```racket
#lang racket

; sleep pauses the execution of the program
; for the amount of time given, in seconds.
;
; int -> void
(define (sleep n)
  ...)

; it's 12:00:00 right now.
(sleep 60)

; it's now 12:01:00.
```

Zzzzzz.....

Maybe we could define a time-period datatype to let callers assign a unit to the time period?

Let's use the template for time-period to figure out how to write snooze.

```
#lang racket

; a time-period is a unit of time, and is one of:
; 'h  : a symbol representing an hour  (60 * 60 seconds)
; 'm  : a symbol representing a minute (60 seconds)
; 's  : a symbol representing a second
; 'ms : a symbol representing a millisecond (1/1000 seconds)

; snooze pauses the execution of the program for the
; amount of time given.
;
; int time-period -> void
(define (snooze n tp)
  (let [(in-seconds (match tp
                      ('h  (* n 60 60))
                      ('m  (* n 60))
                      ('s  n)
                      ('ms (/ n 1000))))]
    (sleep in-seconds)))

(snooze 60 'ms) ; now we know how long we'll snooze for
```

Zzzzzz…..

Here, we used a **symbol**, which is a particular kind of **value**, to indicate what unit of time we are operating on.

```racket
#lang racket

; a time-period is a unit of time, and is one of:
; 'h  : a symbol representing an hour  (60 * 60 seconds)
; 'm  : a symbol representing a minute (60 seconds)
; 's  : a symbol representing a second
; 'ms : a symbol representing a millisecond (1/1000 seconds)

; snooze pauses the execution of the program for the
; amount of time given.
;
; int time-period -> void
(define (snooze n tp)
  (let [(in-seconds (match tp
                      ('h  (* n 60 60))
                      ('m  (* n 60))
                      ('s  n)
                      ('ms (/ n 1000))))]
    (sleep in-seconds)))

(snooze 60 'ms) ; now we know how long we'll snooze for
```

Symbols vs identifiers

A **symbol**, which is a particular kind of **value.** It has meaning within the context of a given program in the language.

An identifier is a **syntactic category** within the context of the language itself. We have seen examples of identifiers:

- language reserved words (keywords): **`lambda`**, **`define`**, **`let`**, …
- bound variables: `(let ((x 31337)) (+ x 42))`

Symbols vs identifiers

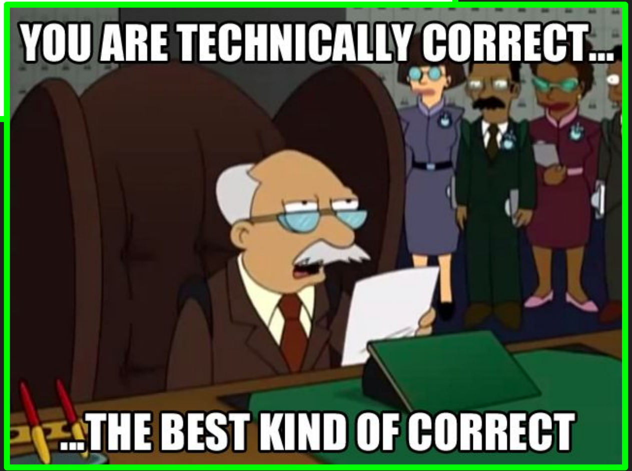
A **symbol**, which is a particular kind of **value.** It has meaning within the context of a given program in the language.

An identifier is a **syntactic category** within the context of the language itself

```
(snooze 60 'ms) ; second argument is a symbol, versus...
(snooze 60 ms)  ; ...a nicer syntax like this??
```

Time unit identifiers, round one

```
; a time-period is a unit of time, and is one of:
; 'h  : an identifier representing an hour  (60 * 60 seconds)
; 'm  : an identifier representing a minute (60 seconds)
; 's  : an identifier representing a second
; 'ms : an identifier representing a millisecond (1/1000 seconds)
(define h 'h)
(define m 'm)
(define s 's)
(define ms 'ms)
```



Time unit identifiers, round one

```
; a time-period is a unit of time, and is one of:
                                          (60 * 60 seconds)
                                         e (60 seconds)
                                         d
                                         second (1/1000 seconds)
```

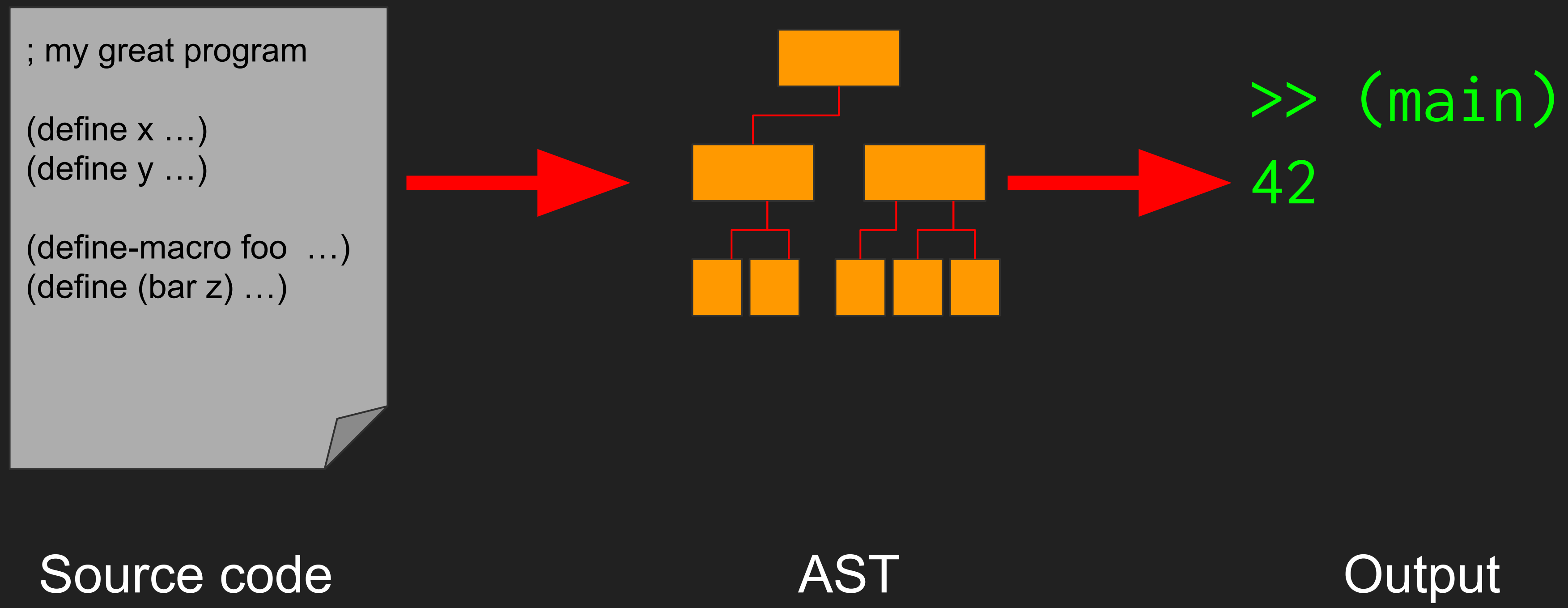
```
(let [(d 31)
      (m "december")
      (y 1974)]
  (... (snooze 60 m))) ; oh no!
```

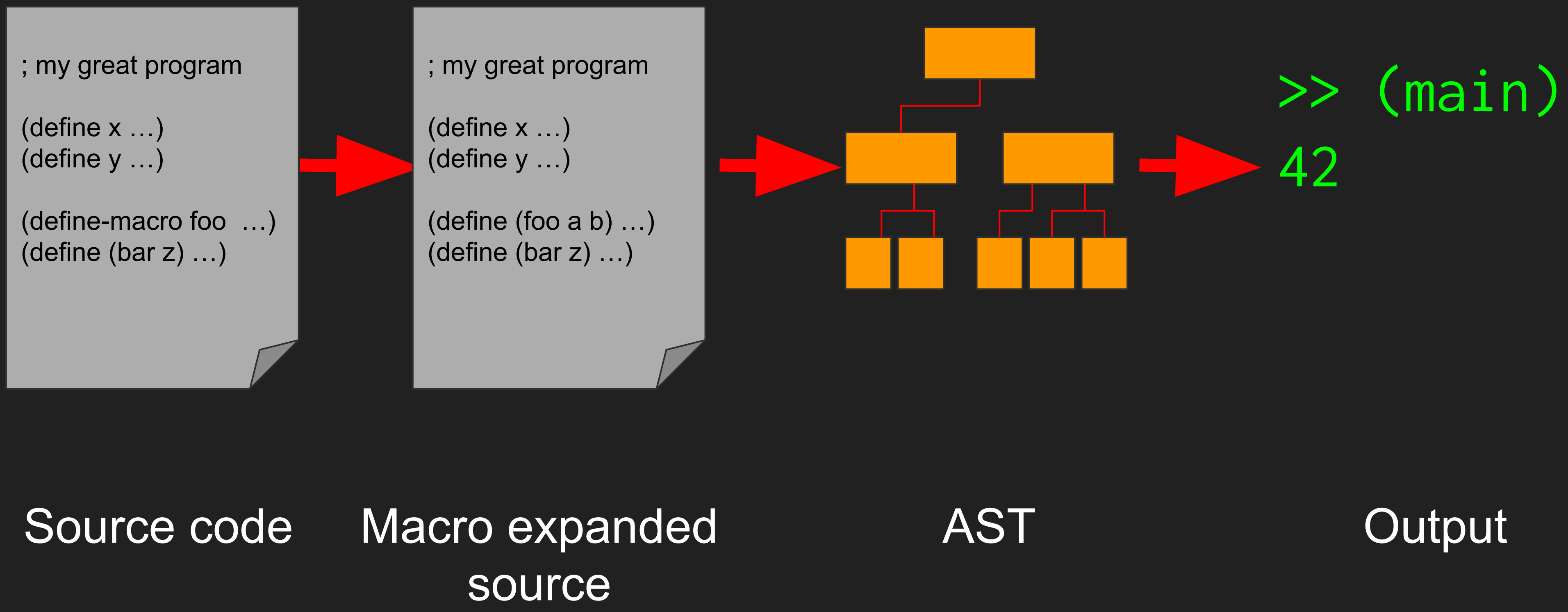
Syntax transformers

Definition: A **syntax transformer** is a special kind of function *that runs at com pile time*, consuming certain code as input and rewriting it as new code.

- Syntax transformers are also known informally as **macros**.
- Syntax transformers are also known as **syntactic forms** or just **forms**

When we talk about "special forms" with special evaluation rules, then, we know that this happens because the **syntax is being transformed to accommodate the special evaluation rules**.


; my great program
(define x …)
(define y …)
(define-macro foo …)
(define (bar z) …)
>> (main)
42
Source code
AST
Output


; my great program
(define x …)
(define y …)
(define-macro foo …)
(define (bar z) …)
; my great program
(define x …)
(define y …)
(define (foo a b) …)
(define (bar z) …)
>> (main)
42
Source code
Macro expanded source
AST
Output

Metaprogramming!!!

Creating **a custom syntax** within the context of a snooze call is our first step into **metaprogramming** a **domain-specific language.**

Our solution to this problem will be to write a **syntax transformer** that transforms the syntax we would like snooze to have, to syntax that Racket already understands

```
(snooze 60 m)    ; to Racket, this will appear as (snooze 60 'm)
(snooze 200 ms) ; to Racket, this will appear as (snooze 200 'ms)
(snooze 60 ns)   ; should emit a syntax error before Racket tries to eval it
```

Syntax transformers

A macro is like **a function that consumes syntax and produces syntax**.

Unlike ordinary functions, which run when they're called during program execution, macros "expand" **at compilation time**, before the final program is compiled and executed.

```
(snooze 60 m)    ; to Racket, this will appear as (snooze 60 'm)
(snooze 200 ms)  ; to Racket, this will appear as (snooze 200 'ms)
(snooze 60 ns)   ; should emit a syntax error before Racket tries to eval it
```

define-syntax-rule

```
;(define-syntax-rule
   (rule-name <pattern>)
   <template>)
```

```
(define-syntax-rule
  (here-is-a-new-function (name arg) body)
  (define (name arg) body))

(here-is-a-new-function (add1 n) (+ 1 n))
```

define-syntax-rule

```
;(define-syntax-rule
   (rule-name <pattern>)
   <template>)

(define-syntax-rule
  (here-is-a-new-function name arg body)
  (define (name arg) body))

(here-is-a-new-function add1 n (+ 1 n))
```

define-syntax-rule vs define-syntax

The define-syntax-rule form binds a macro that matches a single pattern (in this case (snooze n tp)

There is a more general form.

Similar distinction to (define (f x y) …) vs (define f (lambda (x y) …))

```
;(define-syntax-rule (snooze n tp)
  (let* [(in-seconds (match 'tp
                        ('h  (* n 60 60))
                        ('m  (* n 60))
                        ('s  n)
                        ('ms (/ n 1000))))]
    in-seconds))
```

```
(define-syntax snooze
  (syntax-rules ()
    ((snooze n tp)
     (let* [(in-seconds (match 'tp
                           ('h  (* n 60 60))
                           ('m  (* n 60))
                           ('s  n)
                           ('ms (/ n 1000))))]
       in-seconds))))
```

define-syntax-rule vs define-syntax

```
(define-syntax the-function-named
  (syntax-rules (of is)
    [(the-function-named name of arg is body) (define (name arg) body)]))

(the-function-named add1 of n is (+ 1 n))
```

define-syntax-rule

```
(define-syntax-rule (snooze n tp)
  (let* [(in-seconds (match 'tp
                        ('h  (* n 60 60))
                        ('m  (* n 60))
                        ('s  n)
                        ('ms (/ n 1000))))]
    in-seconds))


(let [(m "hello")]
  (snooze 60 m))
```

A macro for list comprehension

```
➜  ~ python3
Python 3.7.6 (default, Dec 30 2019, 19:38:26)
[Clang 11.0.0 (clang-1100.0.33.16)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> [2 * i + 3 for i in range(10)]
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

```
➜  ~ ghci
GHCi, version 8.8.3: https://www.haskell.org/ghc/  :? for help
Loaded package environment from /Users/ntaylor/.ghc/x86_64-darwin-8.8.3/environm
ents/default
Prelude> [2 * i + 3 | i <- [0..9]]
[3,5,7,9,11,13,15,17,19,21]
Prelude>
```

A macro for list comprehension

```
➜  ~ python3
Python 3.7.6 (default, Dec 30 2019, 19:38:26)
[Clang 11.0.0 (clang-1100.0.33.16)] on darwin
Ty
>>
[3
```

```
; Our goal:
(list-comp (+ (* 2 i) 3) : i <- (range 0 10))
```

```
ents/default
Prelude> [2 * i + 3 | i <- [0..9]]
[3,5,7,9,11,13,15,17,19,21]
Prelude>
```

A macro that functions like cond

```
(cond [(= n 1) "one"]
      [(= n 2) "two"]
      [(= n 3) "many"]
      [else "lots"])
```

```
(if (= n 1)
    "one"
    (if (= n 2)
        "two"
        (if (= n 3)
            "many"
            "lots")))
```

A macro that functions like cond

```
(cond [(= n 1) "one"]
      rest ...)

(if (= n 1)
    "one"
    rest ...)
```

A macro that functions like cond

```
(define-syntax my-cond
  (syntax-rules (else)
    [(my-cond [else <val>])
     <val>]
    [(my-cond [<test> <val>] <next> ...)
     (if <test> <val> (my-cond <next> ...))]))

(my-cond [(= n 1) "one"]
         [(= n 2) "two"]
         [(= n 3) "many"]
         [else "lots"])
```

`<rest> ...` says that we match any number of occurrences of a pattern that looks like `<rest>` (that is, are just a single identifier)

A deeper dive into "..."

- On the left hand side of a syntax-rules arm

`<lhs-pattern> ...` says that we match any number of occurrences of a pattern that looks like <lhs-pattern>

- On the right hand side of a syntax-rules arm

`<rhs-pattern> ...` says: for every pattern matched with `<lhs-pattern> ...` , substitute `<lhs-pattern>` with `<rhs-pattern>` .

… "…" behaves a bit like `map` !!!

A deeper dive into "..."

```
#lang racket




; double-all should be a macro that takes a list, and
; returns the list of all elements multiplied by two, quoted.
;
; you would probably not write this as a macro, but we're
; doing so to see how ... works.
(double-all (list 1 (- 3 1) 3))

; should produce
'((* 2 1) (* 2 (- 3 1)) (* 2 3))
```

A deeper dive into "..."

```racket
#lang racket

(define-syntax double-all
  (syntax-rules ()
    [(double-all (list <val> ...)) (list '(* 2 <val>) ...)]))

; double-all should be a macro that takes a list, and
; returns the list of all elements multiplied by two, quoted.
;
; you would probably not write this as a macro, but we're
; doing so to see how ... works.
(double-all (list 1 (- 3 1) 3))

; should produce
'((* 2 1) (* 2 (- 3 1)) (* 2 3))
```

A deeper dive into "..."

```
#lang racket

(define-syntax double-all
  (syntax-rules ()
    [(double-all (list <val> ...)) (list '(* 2 <val>) ...)]))

; double-all should be a macro that takes a list, and
; returns the list of all elements multiplied by two, quoted.
;
; you would probably not write this as a macro, but we're
; doing so to see how ... works.
(double-all (list 1 (- 3 1) 3))

; should produce
'((* 2 1) (* 2 (- 3 1)) (* 2 3))

; Does the macro produce the same thing as the following map?
; Why, or why not? [note: (quasiquote (* 2 (unquote x))) = `(* 2 ,x)]
(map (λ (x) (quasiquote (* 2 (unquote x)))) (list 1 (- 3 1) 3))
```

Macros versus functions

We use functions to manipulate common features from **computation**.

- Functions **evaluate** as the program is evaluated (at run-time)
- Pattern matching deconstructs pieces of data
- Functions operate on arbitrarily-long data by recursing or HOFs like `map`

We use macros to manipulate common features from **syntax**.

- Macros **expand** as the program is parsed (at compile-time)
- Pattern matching deconstructs pieces of code
- Macros operate on arbitrarily-long data by recursing or "…"