Programming Languages, Part B

Status Completed

▼ Lecture 5:

01/19/23

This section starts with an introduction to the Racket programming language. For a more thorough introduction to Racket, consult the first few lectures from the course Systematic Program Design.

Introduction to Racket

- Like SML, Racket focusses on functional programming.
- Unlike SML, there is no static type system and errors do not occur until run-time.
- The DrRacket has built in REPL called the interactions window.

```
; Informs DrRacket to interpret a file as Racket code
#lang racket
; Ensures that all the functions in a file are able to
; be imported to other files
(provide (all-defined-out))
```

Racket definitions, functions, conditionals

- One can use either `first` or `car` to access the head of a list, and use either `rest` or `cdr` to access the tail of a list.
  - The names `car` and `cdr` are is historical accident relating to Lisp, one of Racket's main predecessor languages.

> *The name of the `cons` function is not unreasonable: it is an abbreviation of the word "construct". The origins of the names for `car` and `cdr`, on the other hand, are esoteric: `car` is an acronym from the phrase "Contents of the Address part of the Register"; and `cdr` (pronounced "could-er") is an acronym from the phrase "Contents of the Decrement part of the Register". These phrases refer to the IBM 704 computer on which the original Lisp was developed. -GNU.org*

- It is convention in Racket to use snake case with `-` instead of `_`.

Syntax and (in defense of) parentheses

- Racket has an amazingly simple syntax, such that any term in the language is one of:
  1. An atom like `true`, `34`, or `"hi"`.
  2. A special form like `define`, `lambda`, or `if`.
  3. A sequence like `(t1 t2 ... tn)`.

- Racket syntax is allowed to be so simple because of the use of parentheses.
  - By parenthesizing everything, the program text is converted (essentially parsed) into a tree representing the program.
  - This is trivial and unambiguous, where:
    1. Atoms are leaves.
    2. Sequences are nodes with elements as children.
  - It also removes the need for operator precedence, and makes indentation easy.

```
; For example, there is no need to know what order "3 * 4 + 5" evaluates in
; when programming with Racket


; Multiply first
(+ (* 3 4) 5)

; Add first
(* 3 (+ 4 5))
```

- No one every complains of html, even though it has a very similar nested syntax.

> *"You are entitled to your opinion about syntax, but a good historian wouldn't refuse to study a country where they didn't like people's accents" -Dan Grossman*

Parentheses matter

- In Racket, parentheses are never optional or meaningless.
- In most places, and expression with parentheses `(e)` indicates a function call with zero arguments.

```
;; (1) will be called as a function with no arguments,
;; and therefore an error in thrown

(define (fact_incorrect n)
  (if (= n 0)
      (1)
      (* n (fact_incorrect (- n 1)))))
```

- Because Racket has no static type checker, one needs to write tests for functions in order to catch type errors.

```
; Racket doesn't catch this mistake because it has no type checker
; "*" function will throw a runtime error when (fact_incorrect_d)
; is attempted to be passed in as a number

(define (fact_incorrect_d n)
  (if (= n 0)
```

```
      (1)
      (* n (fact_incorrect_d) (- n 1)))))
```

Dynamic typing

- Racket is a dynamically typed language.
  - Although this means that small errors will not be caught, there are many advantages to dynamic typing.
  - One can build very flexible data structures without having to consult with the type checker that a data structure makes sense.

```
; For example, lists can be made that contain both ints and int lists

(define xs (list (list 4 (list 5 0)) 6 7 (list 8) 9 2))
```

Cond

- `cond` expressions are syntactic sugar for nested `if` `else` expressions.
- For `if` and `cond` statements, anything expression that is not `false` counts as true.

```
; Since "foo" is not false, the if statement will be evaulated to x

(if "foo" x y)
```

Local bindings

- Racket has four different ways to define local variables, all with different semantics.
  - `let`, `let*`, `letrec`, and `define`
  - One should use the the semantics that are most convenient for ones needs.
- `let` expressions can bind to any number of local variables.
  - The expressions are all evaluated using bindings in the environment from before the `let` expression.
- `let*` expressions are evaluated in the environment produced from the previous bindings in the `let*` expression.
- `letrec` expressions are evaluated in the environment that includes all the bindings.
  - This is important for mutually recursive functions.
  - Expressions are still evaluated in order.
- `define` can be used locally, such as at the beginning of function bodies.

Top level bindings

- The top level bindings in a file work like `letrec`.
  - One can refer to both earlier and later bindings.
    - One should only refer to later bindings in function bodies, since function bindings are evaluated after other bindings.

```
; Although b is defined on a line after f,
; it doesn't matter since f has not been evaluated

; f will only be evaluated when it is called

(define (f x) (+ x b))
(define b 3)
```

- One cannot shadow a variable in a module.

```
; x cannot be defined twice

(define x 12)
(define x 97)
```

- Each file in Racket is implicitly a module, where one module can shadow bindings from other modules it uses.
  - This shadowing allows one to redefine/overload any function.

Mutation with set!

- Racket has assignment statements using the keyword `set!` (set bang).
  - `set!` mutates what expression a value is bound to.

```
; Bind 3 to x
(define x 3)
; Change x's binding from 3 to 5
(set! x 5)
```

- `begin` allows the evaluation of a sequence of expressions `(e1 e2 ... en)` in order, where the value of the last expression `en` is produced.
- With `set!` the environment for a closure is still determined when a function is defined, but the body is evaluated when the function is called.
  - Once an expression produces a value, it is irrelevant how the value was produced.
- To avoid `set!` of top-level or built in functions, if a module that defines a value does not `set!` it, then no other modules can `set!` that value.
  - This allows for an easy defense against the mutation of top-level bindings.

The truth about cons

- `cons` really just makes a pair of values (a `cons` cell).
  - By convention, lists are nested pairs that eventually end with `null`.
  - Therefore `car` is equivalent to accessing the first element of the pair, and `cdr` is equivalent to accessing the second element of the pair.
  - A proper list is a series of nested `cons` that ends in null.

    - The list would be improper otherwise.
- Racket allows improper lists because pairs are useful.
- That being said, proper lists with `null` should be used for collections of arbitrary size in order to have a termination condition.
    - The `list?` predicate only returns `true` for proper lists.
    - The `pair?` predicate returns true for anything made by `cons`.

01/20/24

Mcons for mutable pairs

- `mcons` creates a cons cell that is mutable.
    - The list can be mutated with the operators `set-mcar!` and `set-mcdr!`.
    - A proper list cannot be made out of `mcons` cells.

```
; Create a mutable cons cell

(define mlist (mcons 1 (mcons true null)))

; Mutate mlist cdr
; -> (mcons 1 4)
(set-mcdr! mlist 4)
```

Delayed evaluation and thunks

- Understanding when expressions get evaluated semantically is key to understanding many common programming idioms.
- For each language construct, the semantics specify when subexpressions get evaluated.
    - Function arguments are eager, since they are computed first and variables are passed in later.
    - Conditional branches are not eager, since they are evaluated in order.
- Expression evaluation can be delayed by putting it in a function and not calling that function.
    - Thanks to closures, that expression is stored and it can be used later.
    - A zero-argument function used to delay evaluation is called a thunk.

```
; Example of how to thunk an expression in Racket
; The value of x is stored in the closure, and delays the evaluation

(define x 3)
(lambda() (+ x 2))
```

```
; Define a conditional function that takes:
; 1. An expression that evaluates to a boolean
; 2. Two thunks
```

```
(define (my-if x y z)
    (if x (y) (z)))

; Define a factorial function where each conditional branch is a thunk that
; has not been evaluated
; "my-if" only evaluates one of the thunks based on what x evaluates to
(define (fact n)
    (my-if (= n 0)
                (lambda() 1)
          (lambda() (* n (fact (- n 1))))))
```

- It is vital to understand the difference between different semantics.

```
; The 1st case is a normal expression that is evaluated to get a result
; The 2nd case is a thunk where the expression evaluation is delayed

; Evaluate an expression to get a result
e

; A function that, when called, evaluates e and returns the result
(define f (lambda() e))
; Evaluate f to get a thunk, and then call the thunk
(f)
```

Avoiding unnecessary computations

- Thunks allow expensive computations to be skipped if they are not needed.

```
; The thunk may be an expensive computation, therefore it is useful
; to only evaluate it if x evaluates to true

(define (f thunk)
    (if x
        (thunk)
      (...)))

; Pass a thunk into f
(f (lambda() e))
```

- However, calling a thunk that evaluates an expression multiple times can decrease performance.
    - If one encloses the thunk body in a `let` expression, the thunk expression either not evaluated at all, or evaluated once and bound to a value that evaluates immediately.
    - An thunk being remembered after being evaluated once is called lazy evaluation.

```
; The thunk is not evaluated until it is needed
; Then the value of is remembered so future uses of the thunk
; are already completed though lazy evaluation
```

```
(define (f thunk)
    (if x
        (thunk)
      (...)))

; Call f with the thunk wrapped in a let expression
(f (let ([x e]) (lambda() x)))
```

Delay and force

- The `delay` and `force` keywords in Racket are used if it is unknown how many times an expression will be called.
  - If the thunk is never called, it is never evaluated.
  - If the thunk is called once, it is evaluated once.
  - If the thunk is called multiple times, it is evaluated multiple times.
- This programming idiom is commonly referred to as a promise.

```
; The promise is not evaluated until (force promise) is called
; Once the promise has been called, all the subsequent (force promise)'s
; have already been evalauted
(define (f promise)
    (... (if (...) 0 (... (force promise) ...))
             (if (...) 0 (... (force promise) ...))
       ...
       (if (...) 0 (... (force promise) ...))))

; A promise is passed into f, the promise being (delay thunk)
(f (delay (lambda () e)))
```

Using streams

- Streams are functions that *are able* to produce an infinite series of values as needed.
  - They are not infinite, as no real sequence can be infinitely large.
  - Thunks can be used to delay creating most of a sequence if only a shorter prefix of that sequence is needed.
- This is a powerful concept for the division of labor in software systems.
  - The part of the program producing the stream knows how to create values for the stream, but does not know how many values are needed.
  - The part of the program consuming the stream does not know how to create values for the stream, but does know how many values are needed.
- Some examples of a stream is user input, which could ask for a stream of input events as needed.
- We will represent the stream idioms with pairs and thunks.
  - Let a stream be a thunk that when called, returns a pair where the `car` is the evaluation of the previous thunk, and the `cdr` is the next thunk.

```
; The first part is the answer to the previous thunk
; The second part is the next thunk
(next-answer next-thunk)
```

Defining streams

- Every stream is defined in terms of itself.

```
; Define a stream of 1s
; 1 1 1 1 1 ...
(define ones (lambda () (cons 1 ones)))

; Define a stream of integers starting at x
; x (x + 1) (x + 2) ...
(define (f x) (cons x (lambda () (f (+ x 1)))))

; Define a stream of natural numbers starting at 1
; using the more general stream above
(define naturals (lambda () (f 1)))

; Define a stream of powers of 2 starting at 2^0
; 1 2 4 8 16 ...
(define powers-of-two
  (letrec ([f (lambda (x) (cons x (lambda () (f (* x 2)))))])
    (lambda () (f 1))))
```

Memoization

- If a function has no side effects and does not read mutable memory, there is no point computing it twice for the same arguments.
  - Memoization is used to cache previous results.
  - A cache is a data structure used to maintain previous results.
- This is economical if maintaining the cache is cheaper than the recomputing, and the cache results are actually reused.
- Memoization is similar to promises, but for multiple “previous results” for a function that takes arguments.
- Memoization can be used to lead to exponentially faster recursive programs.
  - For functions with more than one recursive call, the results of the first recursive call is stored in the memoization table.
  - The subsequent recursive calls reference that table and compute new values only when they don’t exist yet.
- Memo tables need to be bound *outside* the scope of recursive functions, so they don’t get reset every time a recursive function is called.

Macros: the key points

- A macro definition describes how to transform some new syntax into different syntax in the source language.

    - Macros are one way to implement syntactic sugar.
- Macro systems are languages given to programmers for defining macros.
- Macro expansion is the process of rewriting the syntax for each macro use.
    - Macro expansion happens before runtime, and even before the program is compiled.
- Macros are often overused in places where functions would be better.
    - When in doubt, don't define a macro.

Tokenization, parenthesization, and scope

- Macro systems need to consider how they treat macro expansion of a programming language.
- Macro systems generally work at the level of tokens, not sequences of characters.
    - Therefore, the macro language must know how the programming language tokenizes text.
    - Tokens are usually words, such a variables, function names, numbers, keywords, and so on.
- Macro systems should be elegant enough that they don't require extra parenthesization for safety.
- Macro systems should not shadow variables of the same name, and maintain the correct scope.

Racket macros with define-syntax

- Macros are defined in Racket using the `define-syntax` keyword.
    - `syntax-rules` is used to define other keywords used in the macro.
    - One can define multiple cases in a macro to allow for a more flexible syntax.
    - `...` can be used in Racket to write recursive macros.

```
; Anatomy of a Racket macro
; (define-syntax <macro-name>
;   (syntax-rules (<keywords>...)
;   [<macro_1>
;    <macro_1-expansion>]
;   [<macro_2>
;      <macro_2-expansion>]
;   ...
;   [<macro_n>
;    <macro_n-expansion>]))


; The macro is defined with the name "my-if"
; "then" and "else" are defined as keywords
(define-syntax my-if
    (syntax-rules (then else)
    [(my-if e1 then e2 else e3)
     (if e1 e2 e3)]))
```

Variables, macros, and hygiene

- One should preserve left-to-right evaluation of variables in macros to keep the evaluation order consistent with the macro expansion.
- A macro is hygienic when it is "clean" and keeps macro variables separate from variables in the scope of where the macro is being used.
    1. A programming language should keep macros hygienic by giving macro variables unique names "under the hood".
    2. Programming languages should use lexical scope to look up macro definitions where they are defined, not where they are used.

▼ Lecture 6:

01/21/24

Datatype programming in Racket without structs

- There is no need for datatype bindings in a dynamically typed language, since different primitive types can be mixed anyway using `cons` cells.
- Datatype programming can be implemented in Racket using an `expression -> expression` interpreter.
    - The `car` of a list encodes what kinds of `expression` the datatype is.
- Without the notion of a type system, we need to define for ourselves what an expression is, and how to keep track of it.
    1. First we define our own constructors.

```
;; Helper functions that construct lists where
;; the first element "encodes" the expression
(define (Const i) (list 'Const i))
(define (Negate e) (list 'Negate e))
(define (Add e1 e2) (list 'Add e1 e2))
(define (Multiply e1 e2) (list 'Multiply e1 e2))
```

    2. Next we define our own variant tests.

```
;; Helper functions that test "what kind of expression?"
(define (Const? x) (eq? (car x) 'Const))
(define (Negate? x) (eq? (car x) 'Negate))
(define (Add? x) (eq? (car x) 'Add))
(define (Multiply? x) (eq? (car x) 'Multiply))
```

    3. Then we define our data extraction functions.
        a. This is just better style than using hard-to-read `car` and `cdr`.

```
;; Helper functions that extracts the pieces for "one kind of expression"
(define (Const-int e) (car (cdr e)))
(define (Negate-e e) (car (cdr e)))
(define (Add-e1 e) (car (cdr e)))
(define (Add-e2 e) (car (cdr (cdr e))))
```

```
(define (Multiply-e1 e) (car (cdr e)))
(define (Multiply-e2 e) (car (cdr (cdr e))))
```

4. Finally, we put it all together in a recursive structure without pattern matching.

```
;; This function follows from the same structure used in ML,
;; just without pattern matching
(define (eval-exp e)
  (cond [(Const? e) e]
        [(Negate? e) (Const (- (Const-int (eval-exp (Negate-e e)))))]
        [(Add? e) (let ([v1 (Const-int (eval-exp (Add-e1 e)))]
                        [v2 (Const-int (eval-exp (Add-e2 e)))])
                    (Const (+ v1 v2)))]
        [(Multiply? e) (let ([v1 (Const-int (eval-exp (Multiply-e1 e)))]
                             [v2 (Const-int (eval-exp (Multiply-e2 e)))])
                         (Const (* v1 v2)))]
        [#t (error "eval-exp expected an exp")]))
```

- The `'` at the beginning of text denotes a symbol.
  - Symbols are similar to strings, but can be almost any character sequence.
  - Symbols can be compared using the `eq?` operator.

Datatype programming in Racket with structs

- The keyword `struct` can be used in racket to define structs.

```
;; Define the struct "foo" with the fields "bar", "baz", and "quux"
(struct foo (bar baz quux))

;; Create a new "foo" struct
(foo e1 e2 e3)

;; Find if an expression is a "foo"
(foo? e)

;; Extract fields of the "foo" struct
(define v1 (foo-bar e))
(define v2 (foo-baz e))
(define v3 (foo-quux e))
```

- With structs, we can improve on the previous “datatype” helper functions.

```
;; These structs provide the constructor, tester,
;; and extractor functions all in one line
(struct const (int))
(struct negate (e))
```

```
(struct add (e1 e2))
(struct multiply (e1 e2))
```

- It is important to note that because this a dynamically typed programming language, structs do not use pattern matching.

```
;; This function follows from the same structure as before,
;; except it uses structs instead of the multiple helper functions
(define (eval-exp e)
  (cond [(const? e) e]
        [(negate? e) (const (- (const-int (eval-exp (negate-e e)))))]
        [(add? e) (let ([v1 (const-int (eval-exp (add-e1 e)))]
                        [v2 (const-int (eval-exp (add-e2 e)))])
                    (const (+ v1 v2)))]
        [(multiply? e) (let ([v1 (const-int (eval-exp (multiply-e1 e)))]
                             [v2 (const-int (eval-exp (multiply-e2 e)))])
                         (const (* v1 v2)))]
        [#t (error "eval-exp expected an exp")]))
```

- `#:transparent` is an optional attribute for `struct` definitions.
  - It allows form struct definitions to be printed in the Racket REPL rather than hiding them.
  - This allows for more convenience when debugging programs.

```
; Attributes are tags that can be added on to the end of a struct defintion
(struct foo (field_1 field_2 ...) #:transparent)
```

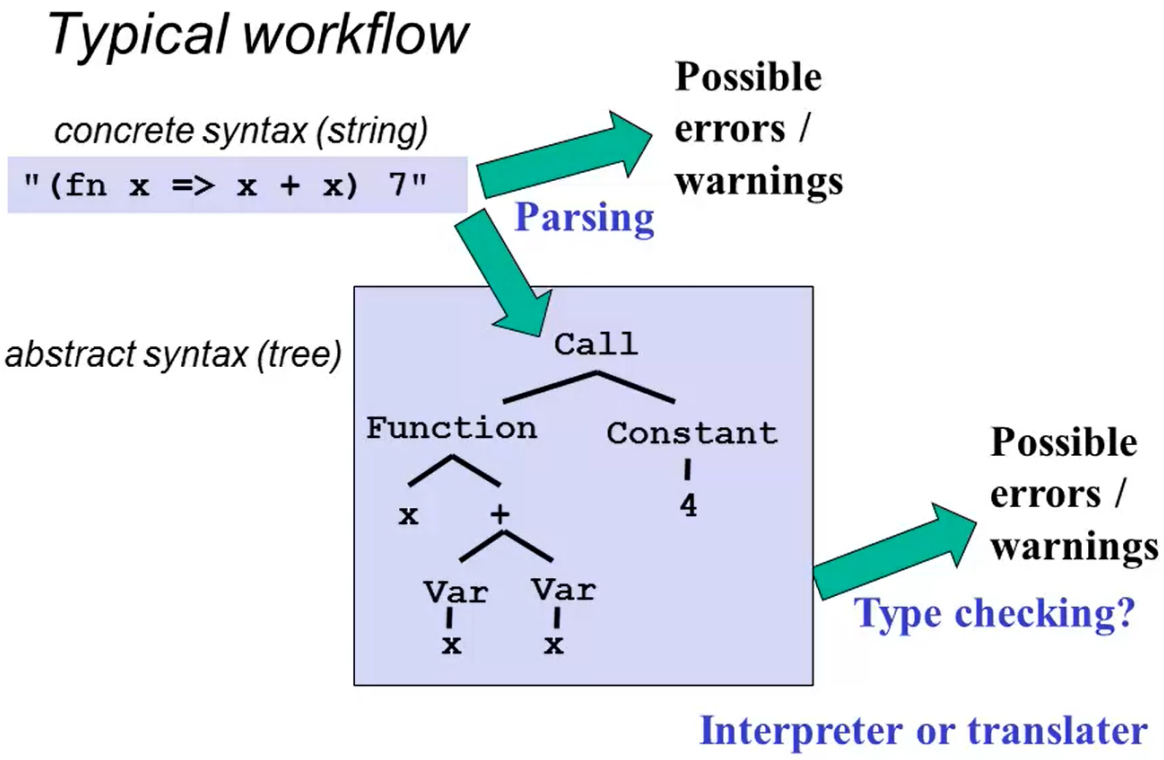
Advantages of structs

- Although structs are able to condense the constructor, tester, and extractor functions into a single function, structs are not syntactic sugar.
- Structs are not lists, as they extend Racket with a new kind of data.
  - Structs are a unique, built in feature that cannot be defined by other language features in Racket.
  - The list approach is error-prone, since is encourages one to think in teams of `car` and `cdr` rather than structs.
  - `list` accessor functions can work on incorrect data elements, whereas `struct` accessor functions do not.
  - Racket has a contract system which lets one check invariants to a struct even if the constructor is exposed to the end user.
    - Properties of a `struct` can be defined so that errors are raised if they are not abided by.

Implementing programming languages

- There is typical workflow when designing a programming language.
  1. Establish a concrete syntax.
  2. The syntax is passed through the parser.
     a. If there are syntax errors, those are raised here.

3. The parser outputs an abstract syntax tree.
4. If the language implement s type checker, the tree will be type checked.
    a. If there are type errors, those are raised here.



- If one programming language A is used to write and interpreter for a second programming language B, programming language A is called the metalanguage.
    - Language A evaluates a program written in the target language B.
- Another approach is to write a compiler.
    - Language A translates a program written in language B into a program written in language C.
- Interpreters and compilers are not mutually exclusive, and are commonly used together in programming language implementations.
    - Which one is used is about the language implementation, not the language definition.
    - There is no such things as a “compiled language” or an “interpreted language”, as programs cannot “see” how the implementation works.
- When implementing language B with language A, one can skip parsing by having programs in language B write abstract syntax trees directly in language A.
    - This isn’t so bad in Racket, as all programs written in it are already in the form of abstract syntax trees.
        - `struct`s are used to define syntax.
        - The interpreter would be a recursive Racket function.

```
;; Let the metalanguage be Racket
;; Let the language implemented be "Arithmetic Language"
;; The interpreter is the eval-exp function

(define (eval-exp e)
  (cond [(const? e) e]
        [(negate? e) (const (- (const-int (eval-exp (negate-e e)))))]
        [(add? e) (let ([v1 (const-int (eval-exp (add-e1 e)))]
                        [v2 (const-int (eval-exp (add-e2 e)))])
                    (const (+ v1 v2)))]
        [(multiply? e) (let ([v1 (const-int (eval-exp (multiply-e1 e)))]
                             [v2 (const-int (eval-exp (multiply-e2 e)))])
                         (const (* v1 v2)))]
        [#t (error "eval-exp expected an exp")]))
```

What your interpreter can and cannot assume

- It is important to consider how an interpreter should check for errors.
- The interpreter must check that the recursive interpreter create a legal syntax tree.
  - The interpreter must check that recursive results are the right kind of value, and throw an error message otherwise.
- An interpreter should always return a value, a kind of expression that evaluate to itself.
  - If it does not return a value, the interpreter has a bug.
- Error detection should be implemented into the interpreter, to detect when a program has a legal AST, but uses the wrong value in the wrong place.

```
; The difference between an interpreter that does not check for
; correct values, and an interpreter that does check


; This interpreter does not implement checking for correct values
[(add? e)
 (let ([i1 (const-int (eval-exp-wrong (add-e1 e)))]
       [i2 (const-int (eval-exp-wrong (add-e2 e)))])
  (const (+ i1 i2)))]

; This intepreter does safeguard for correct values,
; and raises an error if the incorrect value is used
[(add? e)
(let ([v1 (eval-exp (add-e1 e))]
      [v2 (eval-exp (add-e2 e))])
(if (and (const? v1) (const? v2))
    (const (+ (const-int v1) (const-int v2)))
    (error "add applied to non-number")))]
```

Implementing variable and environments

- It is important to consider how an interpreter deals with variables.
- An expression is always evaluated within a current environment.
  - An environment is a mapping from variables → values.
    - This environment can be as simple as a list of pairs.
    - A variable expression can then look up the variable in the environment.
- Most subexpressions use the same environment as the outer expression.
  - The initial environment passed to the interpreter is the empty environment.
  - `let` expressions in Racket would add more pairs to the outer environment to create a new environment.
  - Recursive calls must pass down the correct environment.

Implementing closures

- The most interesting part of implementing programming languages is often implementing closures and the lexical scope for them.
- When defining a function, a closure will be created that includes the environment.
  - Then later when the closure is used, the environment is already stored with it.
  - In a Racket implementation, the function would store the environment in an additional field.
- Closures are values, as they essentially are built from functions and the environment from when the functions are interpreted.
  - This means that functions are not quite values, as a function and an environment are needed to create a value (closure).
- When calling a function:
  1. Use the current environment to evaluate the function as a closure.
     a. Throw an error if the result is a value that is not a closure.
  2. Evaluate the closure's function's body using the closure's environment.
     a. Map the function's argument names to the argument values.
     b. For recursion, extend the environment so that the function's name is mapped to the entire closure.
- Given a closure, the code part is only ever evaluated using the environment part, not the environment at the call-site.

Are closures efficient?

- Implementing closures can be expensive as they copy the entire environment every time a new closure is created.
- This can be remedied by only storing the environment variables that a closure may use.
  - These variables that may be used are called free variables.
  - These are variables that are neither passed in as arguments nor stored in the function body.

Racket functions as "macros" for interpreted languages

- The use of a macro expands into the language syntax before the program is run, and therefore before the main interpreter function is called.

- One can use metalanguage functions that produce the target language syntax using target language “macros”.
  - These macros are just a programming idiom, not an actual feature of the target language.
  - The only job of the macros is to take in syntax, and return syntax.
    - In other words, macros should only produce expressions and never evaluate expressions.

▼ Lecture 7:

01/24/24

ML versus Racket

- The largest difference between the two programming languages is ML’s type system and Racket’s lack of a type system.
  - Without considering syntax, ML is a subset of Racket.
    - ML could do everything Racket can, except ML’s expressions are limited by its type system.

```
;; This function would reach runtime in Racket and throw an error,
;; whereas it would be caught by ML's type checker

(define (f x)
    (+ x (car x)))
```

- However, ML type checking can make functions, which are perfectly valid in Racket, invalid.

```
;; This Racket function can return multiple "types",
;; whereas in ML it would be invalid since a function in ML
;; cannot return multiple types

(define (f x)
    (if (> x 0)
            #t
            (list 1 2 3)))
```

- One way to describe Racket is that it has “one big datatype”, where all values have that type and constructor values are applied implicitly.
  - The Racket interpreter checks the constructor tag and extracts the data from it.
    - Errors are raised when the wrong constructor is used.
  - This is not a perfect analogy, as ML has no support for Racket `struct`s.
    - `struct` updates the “one big datatype” by adding a constructor dynamically during program execution.
      - This is impossible in ML due to its static type system.

What is static checking?

- Static checking is any operation that is done to reject a program after it parses but before it runs.
  - Part of a programming language’s definition is what static checking is performed.

- A common way to define a programming language's static checking is through a type system.
  - When doing so, it is important to distinguish a type system's approach from its purpose.
    - Approach gives each variable, expression, etc. a type.
    - Purposes include to prevent misuse of primitives, enforce abstraction, avoid dynamic checking, and so on.
  - Dynamically typed languages are programming languages that do little, if any, type checking.
  - Language design includes what is checked and how those things are checked.
- Type systems cannot prevent logic or algorithmic errors.
  - This is what examples and tests are for.

Soundness and completeness

- A type system's "correctness" is often defined by both soundness and completeness.
- Assume a type system is supposed to prevent some X:
  - A type system is sound if it never accepts a program that does X (a false negative).

```
;; ML will "never accept" a program that divides an int by a string,
;; among other things
;; Therefore, ML is a sound type system

val x = 4 div "hi"
```

  - A type system is complete if it never rejects a program that will not do X (a false positive).

```
;; ML rejects this program, even through f1 may never be called
;; ML "rejects" a program that will not "call f1"
;; Therefore, ML is not a complete type system

fun f1 x = 4 div "hi"
```

- Most often, the goal for a programming language is to be sound, but not complete.
- It has been proven mathematically that anything checked statically is undecidable.
  - No static checker can do all three of the following:
    1. Always terminate
    2. Be sound
    3. Be complete
- Undecidability is an essential concept at the core of computing, and one could argue that its most important consequence is the inherent approximation of static checking for software development.

Weak typing

- Weak typing denotes a programming language that:
  - Requires static checking, but dynamic checking is optional.

    - So when the program is run, it is allowed to do anything.
- There a many reasons why weak typing may be advantageous.
    - It can make implementing the language easier.
    - It increases performance by not taking time to do dynamic checking.
    - It is lower level and gives the programmer more control.
- This means that as long as a language implementation is correct, the program is allowed to do anything.
- Often times, a language is said to be “static” or “dynamic” based on what its primitive types are allowed to do.
    - In reality, this is not related to static or dynamic checking at all, and is actually a result of the evaluation rules.

Static versus dynamic typing

- While static and dynamic typing are not absolutely better than the other, one can consider arguments for which one is more useful in what situation.

Static typing

- One can assume that data has the expected type without cluttering code with dynamic checks.
- Statically typed languages allow programmers to tag values with constructors only when needed.
- Bugs are caught earlier, as soon as the program is compiled.
- Static typing is “faster”, as storing and checking tags is not needed.
- Modern type systems support reasonable code reuse with features.
- Static types keep functions for multiple types separate, avoiding function/library misuse and confusion.
- Static typing is “better” for prototyping programs, as it essentially “documents” the evolving decisions around data structures, and avoids inconsistent assumptions.
- When changing some code, the type-checker gives a “to-do” list of every other part of the code that needs to be changed.

Dynamic typing

- One can build a list containing many types, or return one of many types.
- Static type checking can forbid programs that do nothing wrong.
- Static type checking only catches implementation bugs, and makes one complacent when catching logic or algorithmic errors.
- Dynamic typing is “faster”, as it can optimize code to remove unnecessary tags and tests.
- One does not need to “code around” type-system limitations.
- With a less restrictive type system, more code can be reused with different data types.
- Dynamic typing is “better” for prototyping programs, avoiding premature commitment to data structures.
- It is easier to change code to be backwards compatible without affecting other parts of the code.

- A better question is “what should be enforced statically?”.
    - There are legitimate trade-offs that one should consider.
    - Does one trust programmers to use flexible programming languages well?