19CSE401 - Compiler Design

Programming Language : Racket

Name	**Roll Number**
Raghul K B	CB.EN.U4CSE19346
O Mahanth	CB.EN.U4CSE19340
V Devakumar	CB.EN.U4CSE19358
V Nithin Krishna	CB.EN.U4CSE19360

Programming Language Description :

Racket is a general-purpose, multi-paradigm programming language and a multi-platform distribution that includes the Racket language, compiler, large standard library, IDE, development tools, and a set of additional languages.The Racket language is a modern dialect of Lisp and a descendant of Scheme.

Multi-Paradigm : Functional, Imperative, Logic, Meta, Modular, Object-Oriented, Reflective

Racket supports multiple threads of control within a program, thread-local storage, some primitive synchronization mechanisms, and a framework for composing synchronization abstractions. The operations in this language are mutable as well as immutable we can use “immutable?” on any operation to check that.

Racket’s synchronization toolbox spans four layers:

- synchronizable events — a general framework for synchronization;
- channels — a primitive that can be used, in principle, to build most other kinds of synchronizable events (except the ones that compose events); and
- semaphores — a simple and especially cheap primitive for synchronization.
- future semaphores — a simple synchronization primitive for use with futures.

Lexical Description:

Racket is a fork of Scheme, the simple language at the core of this course for many years. Scheme was created primarily as an experiment in understanding how programming languages work. Racket retains its basic favor, but it also adds many, many features that make the language useful in the 21st century.

- Upper-case letters (A-Z)
- lower-case letters (a-z)
- Digits (0-9)
- Special symbols - + * / := , . ;. () [] = {} ` white space

. + -
These are used in numbers, and may also occur anywhere in an identifier except as the first character.

( )
Parentheses are used for grouping and to notate lists.

'
The single quote character is used to indicate literal data.

`
The backquote character is used to indicate almost-constant data.

, ,@
The character comma and the sequence comma at-sign are used in conjunction with backquote.

"
The double quote character is used to delimit strings.

\
Backslash is used in the syntax for character constants and as an escape character within string constants.

[ ] { }
Left and right square brackets and curly braces are reserved for possible future extensions to the language.

#
Sharp sign is used for a variety of purposes depending on the character that immediately follows it:

#t #f
These are the boolean constants.

#\
This introduces a character constant.

#(
This introduces a vector constant. Vector constants are terminated by `)' .

#e #i #b #o #d #x
These are used in the notation for numbers.

The following identifiers are syntactic keywords (reserved words):

=>, do, or, and, else, quasiquote, begin, if, quote case, lambda, set!, cond, let, unquote define, let*, unquote-splicing delay, letrec

Syntax Units:

Associativity:
Operations within an infix expression are applied from left to right. So a infix expression like this: 1 + 2 - 3 + 4 - 5 + 6
Corresponds to this Racket expression, where the left-to-right infix operations are converted to inner-to-outer prefix operations: (+ (- (+ (- (+1 2) 3) 4) 5) 6)

Precedence:
Certain operations need to be applied before others, if we have an infix expression like this: 1 + 2 * 3 + 4 * 5 + 6
The * operations have higher precedence than the +. So the expression is evaluated as if it were written like so, and then the operations are applied in the usual left-to-right way: 1 + (2 * 3) + 4 * 5 + 6

In Racket, we would write the original expression like so: (+ (+ (+1 (* 2 3)) (* 4 5) 6)
Subexpressions:

The usual precedence rules can be overridden by using parenthesized subexpressions. In effect, subexpressions are another layer of precedence. Starting with the previous example, we can force the addition operations to happen first by parenthesizing them, and then the remaining operations go left to right:

```
(1+2*(3+4)*(5+6)
```

In Racket, we'd write the expression this way:

```
(* (* (+1 2) (+3 4)) (+5 6))
```

Taking these rules together, our original gnarly example:

```
-1 + 2 -3 * (4+5) /6 ^ 7 + 8 mod 9
```

Would be written in Racket like so:

```
(+ (- (+-1 2) (/ (* 3 (+4 5)) (expt 6 7))) (modulo 8 9))
```

Literals	integer 1 rational 1/2 complex 1+2i floating 3.14 double 6.02e+23 hex #x29 octal #o32 binary #b010101

Arithmetic	+ - * / quotient remainder modulo add1 sub1 max, min, round, floor, ceiling, sqrt, expt, exp, log, sin

Compare	= < <= > >=

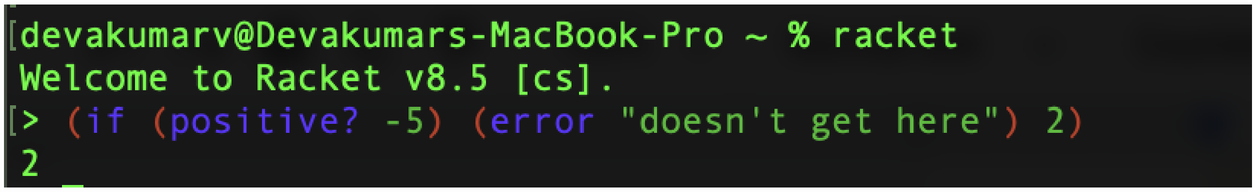
Bitwise	bitwise-ior bitwise-and bitwise-xor bitwise-not arithmetic-shift integer-length

Conditionals: if, cond, and, and or

Evaluates test-expr. If it produces any value other than #f, then then-expr is evaluated, and its results are the result for the if form. Otherwise, else-expr is evaluated, and its results are the result for the if form. The then-expr and else-expr are in tail position with respect to the if form.

```
(if test-expr then-expr else-expr)
```

Example:



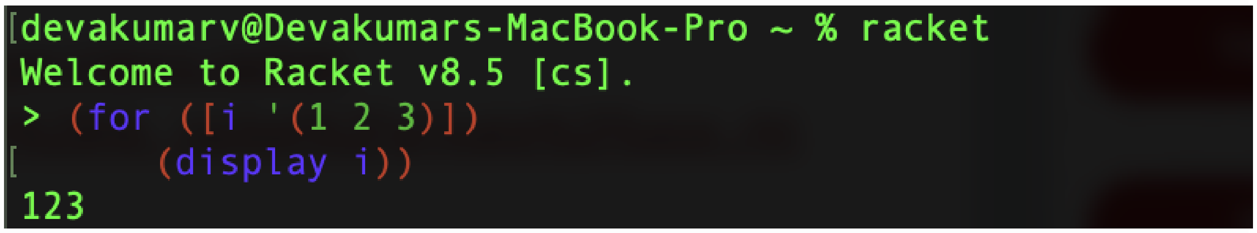
```
[devakumarv@Devakumars-MacBook-Pro ~ % racket
Welcome to Racket v8.5 [cs].
[> (if (positive? -5) (error "doesn't get here") 2)
2
```

Iteration and Comprehension Forms

Iteratively evaluates body's. The for-clauses introduce bindings whose scope includes body and that determine the number of times that body is evaluated. A break-clause either among the for-clauses or bodys stops further iteration.

```
(for (for-clause ...) body-or-break ... body)
```

Example:



```
[devakumarv@Devakumars-MacBook-Pro ~ % racket
Welcome to Racket v8.5 [cs].
> (for ([i '(1 2 3)])
[      (display i))
123
```

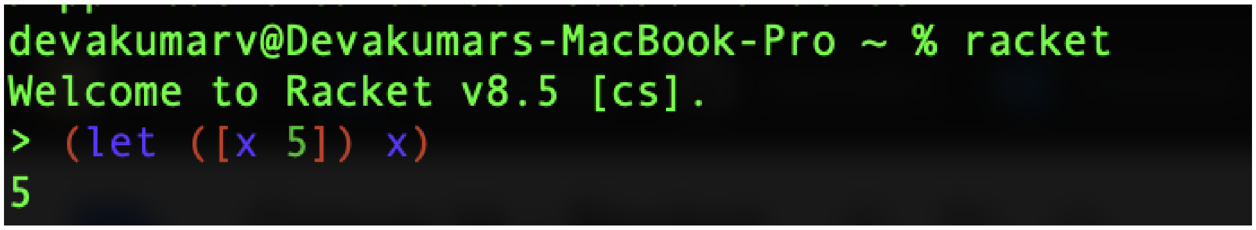
Local Binding: let, let*, letrec, ...

The first form evaluates the val-exprs left-to-right, creates a new location for each id, and places the values into the locations. It then evaluates the bodys, in which the ids are bound. The last body expression is in tail position with respect to the let form. The ids must be distinct according to bound-identifier=?.

```
(let ([id val-expr] ...) body ...+)

(let proc-id ([id init-expr] ...) body ...+)
```

Example:



```
devakumarv@Devakumars-MacBook-Pro ~ % racket
Welcome to Racket v8.5 [cs].
> (let ([x 5]) x)
5
```

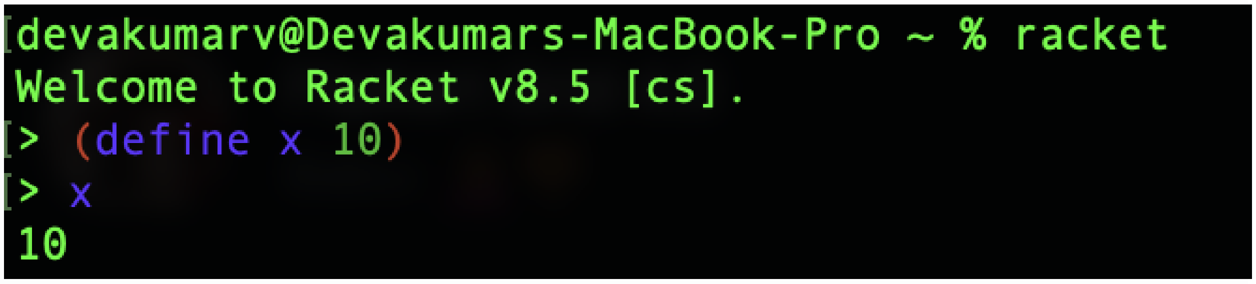
Definitions: define, define-syntax, ...

The first form binds *id* to the result of *expr*, and the second form binds *id* to a procedure. In the second case, the generated procedure is (CVT (*head args*) *body* ...+)

```
(define id expr)

(define (head args) body ...+)
```

Example:



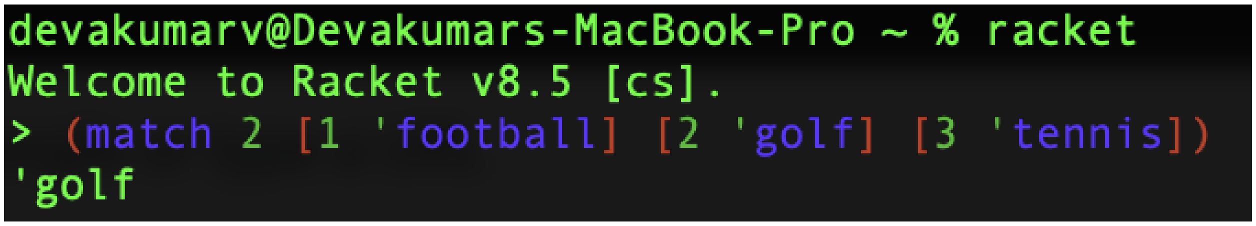
```
[devakumarv@Devakumars-MacBook-Pro ~ % racket
Welcome to Racket v8.5 [cs].
[> (define x 10)
[> x
10
```

Pattern Matching :

The match form takes the result of target-expr and tries to match each pattern in order. As soon as it finds a match, it evaluates the corresponding expr sequence to obtain the result for the match form. If pattern includes pattern variables, they are treated like wildcards, and each variable is bound in the expr to the input fragments that it matched.

```
(match target-expr  [pattern expr ...+] ...)
```

Example :



```
devakumarv@Devakumars-MacBook-Pro ~ % racket
Welcome to Racket v8.5 [cs].
> (match 2 [1 'football] [2 'golf] [3 'tennis])
'golf
```

Classes :

A class specifies a collection of fields; a collection of methods; initial value expressions for the fields; and initialization variables that are bound to initialization arguments.
`(require racket/class)` is the library required to create the class and the objects.

The class system allows a program to define a new class (a derived class) in terms of an existing class (the superclass) using inheritance, overriding, and augmenting:

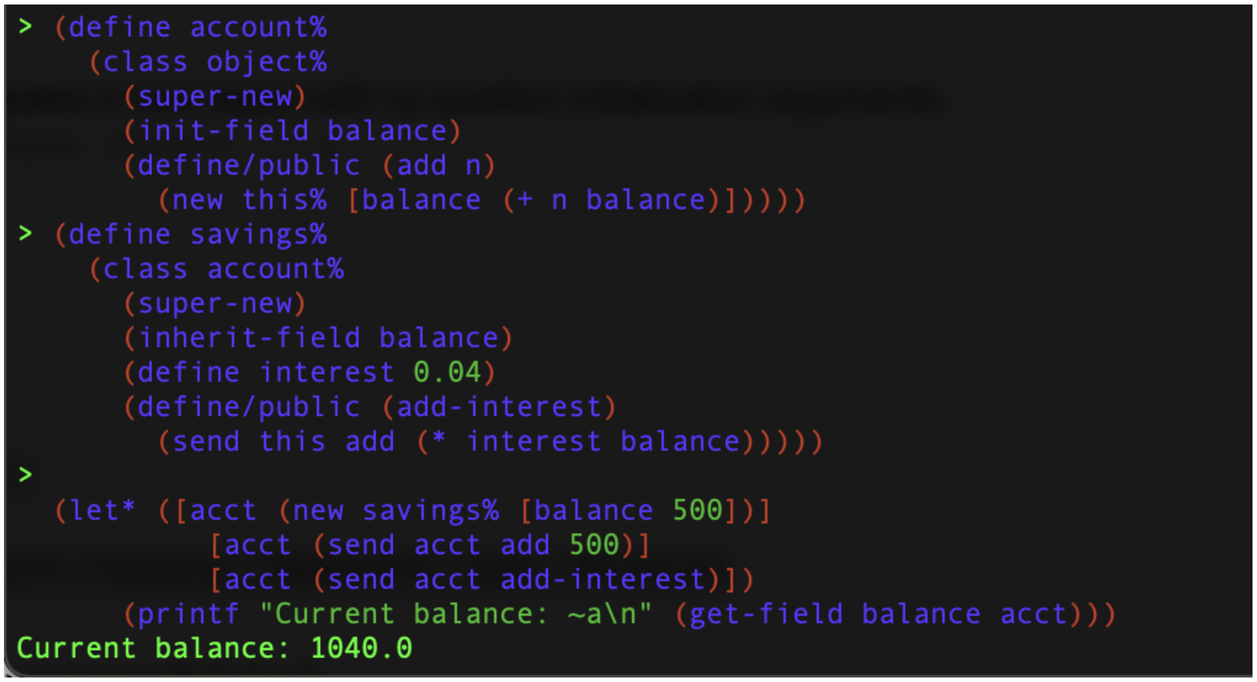
inheritance: An object of a derived class supports methods and instantiates fields declared by the derived class's superclass, as well as methods and fields declared in the derived class expression.

overriding: Some methods declared in a superclass can be replaced in the derived class. References to the overridden method in the superclass use the implementation in the derived class.

augmenting: Some methods declared in a superclass can be merely extended in the derived class. The superclass method specifically delegates to the augmenting method in the derived class.

```
(class superclass-expr class-clause ...)
```

A sample class program for calculating interest and displaying balance :



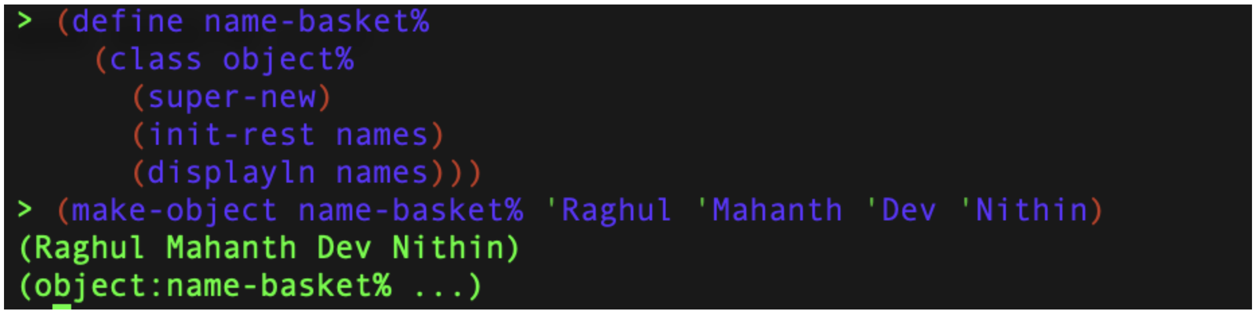
```
> (define account%
    (class object%
      (super-new)
      (init-field balance)
      (define/public (add n)
        (new this% [balance (+ n balance)]))))
> (define savings%
    (class account%
      (super-new)
      (inherit-field balance)
      (define interest 0.04)
      (define/public (add-interest)
        (send this add (* interest balance)))))
>
  (let* ([acct (new savings% [balance 500])]
         [acct (send acct add 500)]
         [acct (send acct add-interest)])
      (printf "Current balance: ~a\n" (get-field balance acct)))
Current balance: 1040.0
```

Objects :

The make-object procedure creates a new object with by-position initialization arguments,

```
        (make-object class init-v ...)
```

Example :



```
> (define name-basket%
    (class object%
      (super-new)
      (init-rest names)
      (displayln names)))
> (make-object name-basket% 'Raghul 'Mahanth 'Dev 'Nithin)
(Raghul Mahanth Dev Nithin)
(object:name-basket% ...)
```

Concurrency and Parallelism :

Threads :

Racket supports multiple threads of evaluation. Threads run concurrently, in the sense that one thread can preempt another without its cooperation, but threads currently all run on the same processor.

`(thread thunk)` is the syntax for creating new threads with no arguments

Suspending, Resuming, and Killing Threads:

`(thread-suspend thd)` Immediately suspends the execution of thd if it is running. If the thread has terminated or is already suspended, thread-suspend has no effect. The thread remains suspended (i.e., it does not execute) until it is resumed with “thread-resume”.

`(thread-resume thd)` Resumes the execution of thd if it is suspended

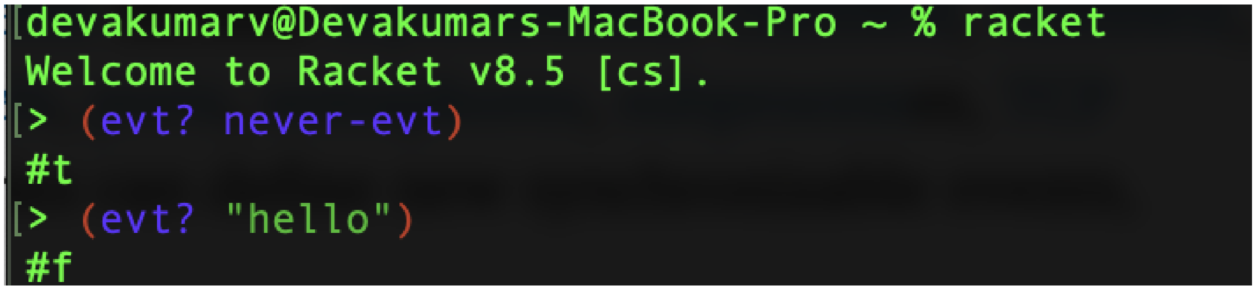
`(kill-thread thd)` Terminates the specified thread immediately, or suspends the thread if thd was created with “thread/suspend-to-kill”. Terminating the main thread exits the application. If thd has already terminated, kill-thread does nothing.

Events :

A synchronizable event (or just event for short) works with the sync procedure to coordinate synchronization among threads.

`(evt? v)` Returns #t (true) if v is a synchronizable event, #f (false) otherwise.

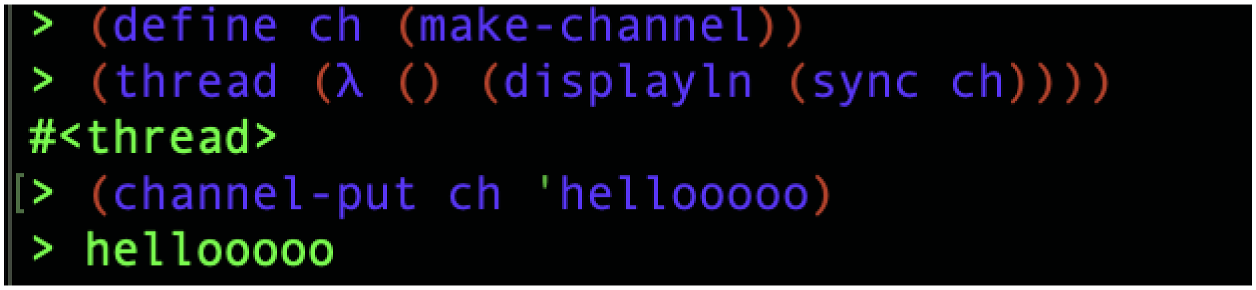
Example :



```
[devakumarv@Devakumars-MacBook-Pro ~ % racket
Welcome to Racket v8.5 [cs].
[> (evt? never-evt)
#t
[> (evt? "hello")
#f
```

`(sync evt ...)` When at least one evt is ready, its synchronization result (often evt itself) is returned. If multiple evts are ready, one of the evts is chosen pseudo-randomly for the result;

Example:



```
> (define ch (make-channel))
> (thread (λ () (displayln (sync ch))))
#<thread>
[> (channel-put ch 'hellooooo)
> hellooooo
```

Channel :
A channel both synchronizes a pair of threads and passes a value from one to the other. Channels are synchronous; both the sender and the receiver must block until the (atomic) transaction is complete.

`(make-channel)` is used to create a new channel

Semaphores :
`(make-semaphore [init])` Creates and returns a new semaphore with the counter initially set to init. If init is larger than a semaphore's maximum internal counter value, the exn:fail exception is raised.

`(semaphore-post sema)` Increments the semaphore's internal counter and returns #<void>. If the semaphore's internal counter has already reached its maximum value, the exn:fail exception is raised.

`(semaphore-wait sema)` Blocks until the internal counter for semaphore sema is non-zero. When the counter is non-zero, it is decremented and semaphore-wait returns void.

Future Semaphores :
A future semaphore is similar to a plain semaphore, but future-semaphore operations can be performed safely in parallel (to synchronize parallel computations). In contrast, operations on plain semaphores are not safe to perform in parallel, and they therefore prevent a computation from continuing in parallel.

`(make-fsemaphore init)` Creates and returns a new future semaphore with the counter initially set to init.

`(fsemaphore-post fsema)` Increments the future semaphore's internal counter and returns void.

`(fsemaphore-wait fsema)` Blocks until the internal counter for fsema is non-zero. When the counter is non-zero, it is decremented and fsemaphore-wait returns void.

Semantics:

To summarize, if Typed Racket can determine the type a variable must have based on a predicate check in a conditional expression, it can narrow the type of the variable within the appropriate branch of the conditional. It supports narrowing as the bigger primitive type value is assigned to a smaller primitive type value.

Basic Types:

The most basic types in Typed Racket are those for primitive data, such as True and False for booleans, String for strings, and Char for characters.

```
> '"hello, world"
- : String

"hello, world"

> #\f
```

Function Types:

Function types are constructed using ->, where the last type is the result type and the others are the argument types. Here are some example function types:

```
(-> Number Number)
(-> String String Boolean)
(-> Char (Values String Natural))

> (lambda ([x : Number]) x)
- : (-> Number Number)
```

#<procedure>

Types for Functions with Optional or Keyword Arguments
Racket functions often take optional or keyword arguments in addition to standard mandatory arguments.

```
(->* () (Number) Number)
(->* (String String) Boolean)
(->* (#:x Number) (#:y Number) (values Number Number))
```

Union Types:
Sometimes a value can be one of several types. To specify this, we can use a union type, written with the type constructor U.

```
> (let ([a-number 37])
    (if (even? a-number)
        'yes
        'no))
- : (U 'no 'yes)

'no
```

Recursive Types:
Recursive types are types whose definitions refer to themselves. This allows a type to describe an infinite family of data.

Structure Types:
Using struct introduces new types, distinct from any previous type.

```
(struct point ([x : Real] [y : Real]))
```

Subtyping:
In Typed Racket, all types are placed in a hierarchy, based on what values are included in the type.

```
(: f (-> Real Real))
(define (f x) (* x 0.75))

(: x Integer)
(define x -125)

(f x)
```

References :

1) https://docs.racket-lang.org/

2) http://web.mit.edu/racket_v612/amd64_ubuntu1404/racket/doc/reference/generic-numbers.html#%28def._%28%28quote._~23~25kernel%29._floor%29%29

3) https://plt.cs.northwestern.edu/pkg-build/doc/index.html