Today we will...

1. Learn the first steps of implementing a language
2. Design an interpreter of arithmetic operations
3. Handling operations with multiple arguments

Our goal is to implement an evaluation function that takes an expression and yields a value.

\[
\begin{align*}
\text{expression} &= \text{value} \mid \text{variable} \mid \text{function-call} \\
\text{value} &= \text{number} \\
\text{function-call} &= ( \text{expression} )
\end{align*}
\]
How do we evaluate an expression

What is an expression?

expression = value | variable | function-call

How do we evaluate a value?
How do we evaluate an expression

What is an expression?

\[ \text{expression} = \text{value} | \text{variable} | \text{function-call} \]

- How do we evaluate a value? The evaluation of a value \( \mathbf{v} \) is \( \mathbf{v} \) itself.

\[
\text{check-equal? 10 (eval-exp (r:number 10))}
\]

- How do we evaluate a function call?
How do we evaluate an expression

What is an expression?

expression = value | variable | function-call

How do we evaluate a value? **The evaluation of a value v is v itself.**

(check-equal? 10 (eval-exp (r:number 10)))

How do we evaluate a function call? **The evaluation of a function call evaluates each expression from left to right and then it applies the function to the arguments.**
Example

How do we evaluate a function call? The evaluation of a function call evaluates each expression from left to right and then it applies the function to the arguments.

\[
\text{(eval-exp '(- (+ 3 2) (* 5 2)))}
\]
Example

How do we evaluate a function call? The evaluation of a function call evaluates each expression from left to right and then it applies the function to the arguments.

```
(eval-exp
  (-
    (+ 3 2)
    (* 5 2)))
```

= ((eval-exp ' -
    (eval-exp '( + 3 2))
    (eval-exp '(* 5 2)))

①
← evaluate ' -
← evaluate '( + 3 2)
← evaluate '(* 5 2)

②
← evaluate ' +, evaluate 3, evaluate 2
← evaluate ' * , evaluate 5, evaluate 2
Example

How do we evaluate a function call? The evaluation of a function call evaluates each expression from left to right and then it applies the function to the arguments.

```
(eval-exp
  '(#
    (+ 3 2)
    (* 5 2)))
```

= ((eval-exp ')
   (eval-exp '(+ 3 2))
   (eval-exp '(* 5 2)))

= ((eval-exp ')
   (eval-exp '+ 3 2)
   (eval-exp '* 5 2))

①
← evaluate '->
← evaluate '(+ 3 2)
← evaluate '(* 5 2)

②
← evaluate '+, evaluate 3, evaluate 2
← evaluate '*', evaluate 5, evaluate 2

③
← numbers are values, so just return those
← numbers are values, so just return those
How do we evaluate arithmetic operators?

\[
\begin{align*}
= & \quad (\text{eval-exp } \text{'}-\text{'}) \\
& \quad ((\text{eval-exp } \text{'}+\text{'}) \\ & \quad 3 \ 2) \\
& \quad ((\text{eval-exp } \text{'}*\text{'}) \\ & \quad 5 \ 2))
\end{align*}
\]
How do we evaluate arithmetic operators?

\[
= ((\text{eval-exp} \ '-' )
 (\text{eval-exp} \ '+' 3 2)
 (\text{eval-exp} \ '*' 5 2))
\]

\[
= (\text{Evaluate } '-' \text{ as function } -
 (\text{Evaluate } '+' \text{ as function } +
 (\text{Evaluate } '*' \text{ as function } *))
\]
Evaluation of arithmetic expressions

1. When evaluating a number, just return that number
2. When evaluating an arithmetic symbol, return the respective arithmetic function
3. When evaluating a function call evaluate each expression and apply the first expression to remaining ones

Essentially evaluating an expression translates our AST nodes as a Racket expression.
Implementing eval-exp...
Specifying `eval-exp`

- We are use the AST we defined in Lesson 5, not datums.
- Assume function calls are binary.

```scheme
(check-equal? (r:eval-exp (r:number 5)) 5)
(check-equal? (r:eval-exp (r:number 10)) 10)
(check-equal? (r:eval-exp (r:variable? '+)) +)
(check-equal? (r:eval-exp (r:apply (r:variable '+) (list (r:number 10) (r:number 5)))) 15)
```
Implementing eval-exp

We are using the AST we defined in Lesson 5, not datums. Assume function calls are binary.

```
(define (r:eval-exp exp)
  (cond
   ; 1. When evaluating a number, just return that number
   [(r:number? exp) (r:number-value exp)]
   ; 2. When evaluating an arithmetic symbol, return the respective arithmetic function
   [(r:variable? exp) (r:eval-builtin (r:variable-name exp))]
   ; 3. When evaluating a function call evaluate each expression and apply the first expression to remaining ones
   [(r:apply? exp)
    ((r:eval-exp (r:apply-func exp))
     (r:eval-exp (first (r:apply-args exp)))
     (r:eval-exp (second (r:apply-args exp)))]
   [else (error "Unknown expression:" exp)])
```

Implementing `r:eval-builtin`

Spec

```
(check-equal? (r:eval-builtin '+') +)
(check-equal? (r:eval-builtin '-') -)
(check-equal? (r:eval-builtin '/') /)
(check-equal? (r:eval-builtin '*') *)
(check-equal? (r:eval-builtin 'foo') #f)
```
Implementing `r:eval-builtin`

Spec

```scheme
(check-equal? (r:eval-builtin '+) +)
(check-equal? (r:eval-builtin '-) -)
(check-equal? (r:eval-builtin '/) /)
(check-equal? (r:eval-builtin '*') *)
(check-equal? (r:eval-builtin 'foo) #f)
```

Solution

```scheme
(define (r:eval-builtin sym)
  (cond
    [(equal? sym '+) +]
    [(equal? sym '*) *]
    [(equal? sym '-) -]
    [(equal? sym '/') /]
    [else #f]))
```
Handling functions with an arbitrary number of parameters

(required for Homework 3)
Function **apply**

Function \((\text{apply } f \text{ args})\) applies function \(f\) to the list of arguments \(\text{args}\).

**Examples**

\[
\text{(check-equal? (apply + (list 1 2 3 4)) 10)}
\]

**Example:** Implement \((\text{sum } l)\) that takes returns the summation of all members in \(l\) using apply.

**Spec**

\[
\text{(check-equal? (sum (list)) 0)}
\]

\[
\text{(check-equal? (sum (list 1 2 3 4)) 10)}
\]
Function **apply**

Function `(apply f args)` applies function `f` to the list of arguments `args`.

**Examples**

```scheme
(check-equal? (apply + (list 1 2 3 4)) 10)
```

**Example:** implement `(sum l)` that takes returns the summation of all members in `l` using `apply`.

**Spec**

```scheme
(check-equal? (sum (list)) 0)
(check-equal? (sum (list 1 2 3 4)) 10)
```

**Solution**

```scheme
(define (sum l) (apply + l))
```
Handling multiple-args without apply

Some multi-arg operations can be implemented without the need of `apply`.

Implement \((\text{sum} \; 1)\) without using `apply`.

Spec

\[
\begin{align*}
\text{(check-equal? (sum (list)) 0)} \\
\text{(check-equal? (sum (list 1 2 3 4)) 10)}
\end{align*}
\]
Handling multiple-args without apply

Some multi-arg operations can be implemented without the need of \texttt{apply}.

Implement \((\texttt{sum } 1)\) without using \texttt{apply}.

Spec

\[
\begin{align*}
& (\texttt{check-equal? } (\texttt{sum } \texttt{(list)}) \texttt{ 0}) \\
& (\texttt{check-equal? } (\texttt{sum } \texttt{(list } 1 \texttt{ 2 } 3 \texttt{ 4)}) \texttt{ 10})
\end{align*}
\]

Solution 1

\[
(\texttt{define } (\texttt{sum } 1)
(\texttt{cond}
  [(\texttt{empty? } 1) \texttt{ 0}]
  [\texttt{else } (+ (\texttt{first } 1) (\texttt{sum } \texttt{(rest } 1))))]))
\]

Solution 2 (foldl is tail-recursive)
Handling multiple-args without apply

Some multi-arg operations can be implemented without the need of `apply`.

Implement `(sum l)` without using `apply`.

Spec

\[
\begin{align*}
&\text{(check-equal? (sum (list)) 0)} \\
&\text{(check-equal? (sum (list 1 2 3 4)) 10)}
\end{align*}
\]

Solution 1

\[
\begin{align*}
&\text{(define (sum l)} \\
&\text{ (cond} \\
&\text{ [ (empty? l) 0]} \\
&\text{ [ else (+ (first l) (sum (rest l)))]])}
\end{align*}
\]

Solution 2 (foldl is tail-recursive)

\[
\begin{align*}
&\text{(define (sum l) (foldl + 0 l))}
\end{align*}
\]
Implementing functions with multi-args

How could we implement a function with multiple parameters, similar to `+`? **Use the . notation.**

The dot . notation declares that the next variable represents a list of zero or more parameters.

**Examples**

```scheme
(define (map-ex f . args)
  (map f args))

(check-equal? (list 2 3 4) (map-ex (curry + 1) 1 2 3))

(define (sum . l) (foldl + 0 l))
(check-equal? 6 (sum 1 2 3))
```
Can we implement `apply`?

Is there any way we can implement `apply` in terms of other functions we have learned before?
Can we implement apply?

Is there any way we can implement apply in terms of other functions we have learned before?

Yes!

Solution

```racket
#lang racket

(define (apply f l)
  (define (on-elem elem new-f)
    (new-f elem))
  (foldl on-elem (curry f) l))

; Test case
(require rackunit)
(define (f a b c d) (list a b c d))
(check-equal? (list 1 2 3 4) (apply f (list 1 2 3 4)))
```
Modules
Modules

- Modules encapsulate a unit of functionality
- A module groups a set of constants and functions
- A module encapsulates (hides) auxiliary top-level functions
- Each file represents a module
Modules in Racket

Each file represents a module. A bindings becomes visible through the `provide` construct. Function `(require "filename")` loads a module

- `(provide (all-defined-out))` makes all bindings visible
- `(provide a c)` makes binding a and c visible
- `(require "foo.rkt")` makes all bindings of the module in file `foo.rkt` visible in the current module. Both files have to be in the same directory.

File: foo.rkt

```racket
#lang racket
; Make variables a and c visible
(provide a c)
(define a 10)
(define b (+ a 30))
(define (c x) b)
```

File: main.rkt

```racket
(require "foo.rkt")
(c a)
; b is not visible
```