CS450

Structure of Higher Level Languages

Lecture 09: Building the AST / Tail recursion

Tiago Cogumbreiro

Building the AST

The AST of values

value = *number* | *void* | *func-dec*
func-dec = `(lambda (` *variable** `)` *term+* `)`

Implementation

```
(define (r:value? v)
  (or (r:number? v)
      (r:void? v)
      (r:lambda? v)))
(struct r:void () #:transparent)
(struct r:number (value) #:transparent)
(struct r:lambda (params body) #:transparent)
```

How do we represent?

1. `10`
2. `(void)`
3. `(lambda () 10)`

AST

The AST of values

```
value = number | void | func-dec
func-dec = (lambda ( variable* ) term+ )
```

Implementation

```
(define (r:value? v)
  (or (r:number? v)
      (r:void? v)
      (r:lambda? v)))
(struct r:void () #:transparent)
(struct r:number (value) #:transparent)
(struct r:lambda (params body) #:transparent)
```

How do we represent?

1. `10`
2. `(void)`
3. `(lambda () 10)`

AST

```
(r:number 10)     ; <- 1
(r:void)          ; <- 2
(r:lambda (list)  ; <- 3
  (list (r:number 10)))
```

The AST of expressions

```
expression = value | variable | apply
apply = ( expression+ )
```

Implementation

```
(define (r:expression? e)
  (or (r:value? e)
      (r:variable? e)
      (r:apply? e)))
(struct r:variable (name) #:transparent)
(struct r:apply (func args) #:transparent)
```

How do we represent?

1. `x`
2. `(f 10)`

AST

The AST of expressions

expression = *value* | *variable* | *apply*
apply = `(` *expression*+ `)`

Implementation

```
(define (r:expression? e)
  (or (r:value? e)
      (r:variable? e)
      (r:apply? e)))
(struct r:variable (name) #:transparent)
(struct r:apply (func args) #:transparent)
```

How do we represent?

1. `x`
2. `(f 10)`

AST

```
; 1:
(r:variable 'x)
; 2:
(r:apply
  (r:variable 'f)
  (list (r:number 10)))
```

UMass Boston

The AST of terms

term = *define* | *expression*
define = `( define` *identifier* *expression* `)` | `( define` `(` *variable+* `)` *term+*`)`

```
(define (r:term? t)
  (or (r:define? t)
      (r:expression? t)))
(struct r:define (var body) #:transparent)
```

Which Racket code is this?

```
(r:define (r:variable 'f)
  (r:lambda (list (r:variable 'y))
    (list
      (r:apply (r:variable '+)
               (list (r:variable 'y) (r:number 10))))))
```

The AST of terms

term = *define* | *expression*
define = `( define` *identifier* *expression* `)` | `( define` `(` *variable+* `)` *term+*`)`

```
(define (r:term? t)
  (or (r:define? t)
      (r:expression? t)))
(struct r:define (var body) #:transparent)
```

Which Racket code is this?

```
(r:define (r:variable 'f)
  (r:lambda (list (r:variable 'y))
    (list
      (r:apply (r:variable '+)
               (list (r:variable 'y) (r:number 10)))))))
```

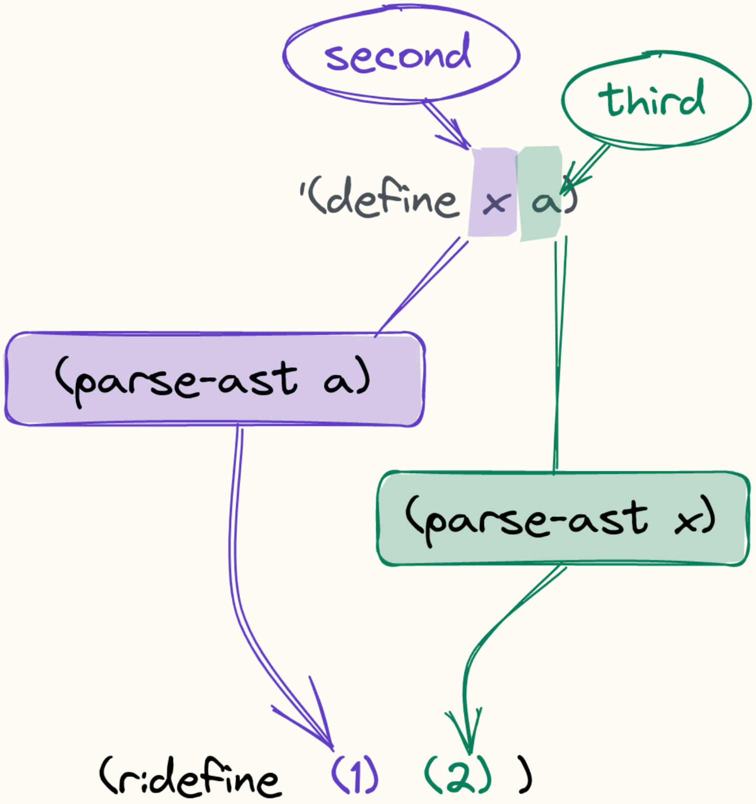
Answer 1

```
(define (f y) (+ y 10))
```

Answer 2

```
(define f
  (lambda (y) (+ y 10)))
```

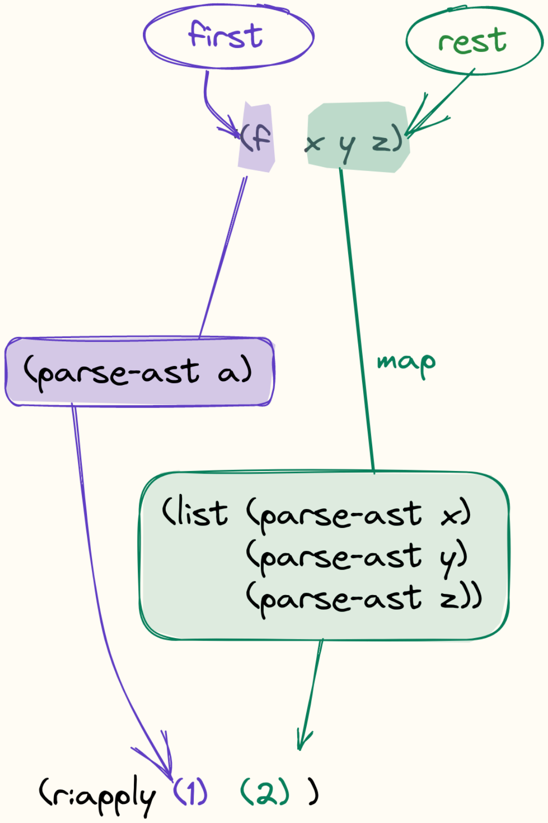
How to implement make-define-basic?



In `make-define-basic`, given a quoted lambda in `node`

1. convert the name of the variable `x` using `parse-ast`
2. convert the body using `parse-ast`
3. return an `r:define`

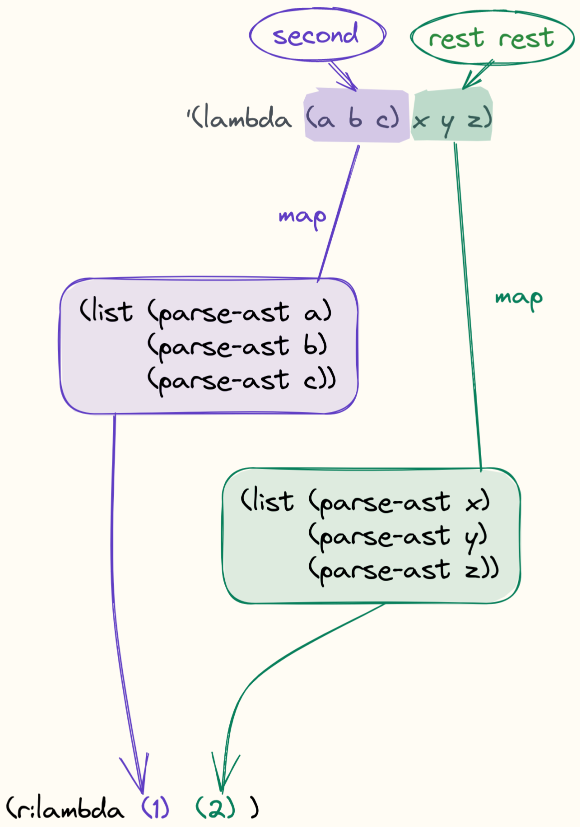
How to implement make-apply?



In `make-apply`, given a quoted lambda in `node`

1. convert the function using `parse-ast`
2. convert the args: go from list of quoted expressions to list of expressions
3. return an `r:apply`

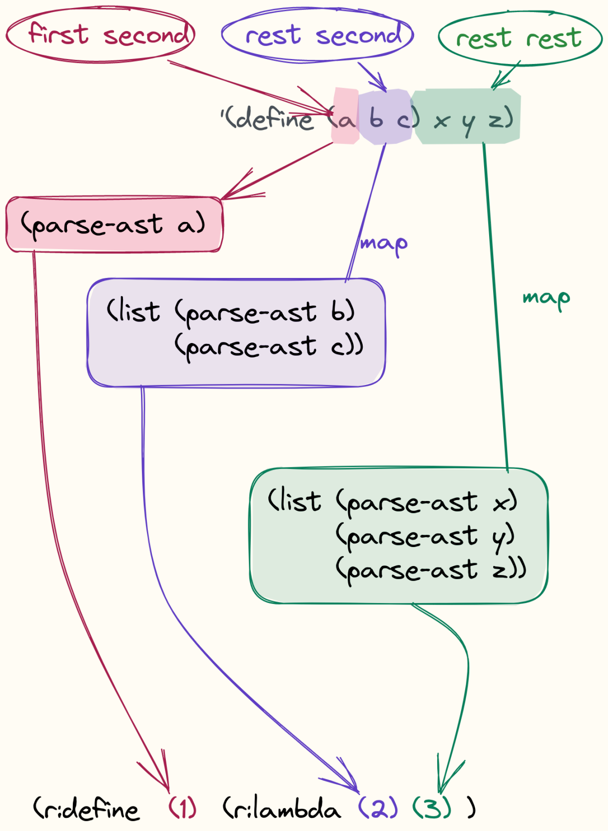
How to implement make-lambda?



In `make-lambda`, given a quoted lambda in `node`

1. convert the parameters: go from list of symbols to list of variables
2. convert the body: go from list of quoted terms to list of terms
3. return an `r:lambda`

How to implement make-define-func?



In `make-define-func`, given a quoted lambda in `node`

1. convert the name of the function with `parse-ast`
2. convert the parameters with `parse-ast`: go from list of symbols to list of variables
3. convert the body with `parse-ast`: go from list of quoted terms to list of terms
4. return an `r:define` that holds an `r:lambda`

Tail-call optimization

max: attempt 1

```
(define (max l)
  (match l
    [(list) (error "max: expecting a non-empty list!")]
    [(list x) x]                                   ; The list only has one element (the max)
    [(list h l ...) #:when (> h (max l)) h] ; The max of the rest is smaller than 1st
    [(list _ l ...) (max l)]))              ; Otherwise, use the max of the rest
```

max: attempt 2

We use a local variable to cache a duplicate computation.

```
(define (max2 x y) (if (> x y) x y))
(define (max l)
  (match l
    [(list) (error "max: expecting a non-empty list!")]
    [(list h) h]
    [(list h l ...)
      (define rest-max (max l)) ; Cache the max of the rest
      (max2 h rest-max)]))
```

- Attempt #1: 20 elements in 75.78ms
- Attempt #2: 1,000,000 elements in 101.15ms

5000× more elements for the same amount of time!

UMass Boston

Can we do better?

max: attempt 3

```
(define (max l) =
  ; 1. Abstract the maximum between two numbers
  (define (max2 x y) (cond [(< x y) y] [else x]))
  ; 2. Use parameters to store accumulated results
  (define (max-aux curr-max l)
    ; 3. Accumulate maximum number before recursion
    (match l
      [(list) curr-max]
      [(list h l ...)
        (define new-max (max2 curr-max h))
        (max-aux new-max l)])) ; Otherwise, recurse
  (match l
    [(list) (error "max: empty list")] ; 4. Only test if the list is empty once
    [(list h l ...) (max-aux h l)]))
```

Comparing both attempts

	Element count	Execution time	Increase
Attempt #2	1,000,000	101.15ms	
Attempt #3	1,000,000	20.98ms	4.8× speedup
Attempt #2	10,000,000	1410.06ms	
Attempt #3	10,000,000	237.66ms	5.9× speedup

> Why is attempt #3 so much faster?

Because attempt #3 is being target of a Tail-Call optimization!

UMass Boston

How are both attempts different?

Recursive step

Attempt 2

```
(define rest-max (max l))
(max2 h rest-max)]))
```

Attempt 3

```
(define new-max (max2 curr-max h))
(max-aux new-max l)
```

Recursive step (simplified)

Attempt 2 (recursive call first)

```
(max2 h (max l))
```

Attempt 3 (recursive call last)

```
(max-aux (max2 curr-max h) l)
```

Tail-call optimization

Why does it work?

Call stack & Activation frame

- **Call Stack:** To be able to call and return from functions, a program internally maintains a stack called the ***call-stack***, each of which holds the execution state at the point of call.
- **Activation Frame:** An activation frame maintains the execution state of a running function. That is, the activation frame represents the local state of a function, it holds the state of each variable.
- **Push:** When calling a function, the caller creates an activation frame that is used by the called function (eg, to pass arguments to the function being called).
- **Pop:** Before a function returns, it pops the call stack, freeing its local state.

Consider executing the factorial

Program

```
(define (fact n)
  (cond
    [(= n 1) 1]
    [else
     (* n (fact (- n 1)))]))
```

Evaluation

```
(fact 3)
(* 3 (fact 2))
(* 3 (* 2 (fact 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

Call-Stack

```
[n=3,return=(* 3 (fact 2))]
[n=3,return=(* 3 ?)],[n=2,return=(* 2 (fact 1))]
[n=3,return=(* 3 ?)],[n=2,return=(* 2 ?)],[n=1,return=1]
[n=3,return=(* 3 ?)],[n=2,return=2]
[n=3,return=6]
```

Call-stack and recursive functions

> Recursive functions pose a problem to this execution model, as **the call-stack may grow unbounded**! Thus, most non-functional programming languages are conservative on growing the call stack.

```
def fact(n):
  return 1 if n <= 1 else n * fact(n - 1)
fact(1000)
```

Outputs

```
  File "<stdin>", line 1, in fact
RuntimeError: maximum recursion depth exceeded
```

Factorial: attempt #2

Program

```
(define (fact n)
  (define (fact-iter n acc)
    (cond
      [(= n 0) acc]
      [else
        (fact-iter (- n 1) (* acc n)) ]))
  (fact-iter n 1))
(fact 3)
```

Evaluation

```
(fact 3)
(fact-iter 3 1)
(fact-iter 2 3)
(fact-iter 1 6)
6
```

Factorial: attempt #2

Call stack

```
[n=3,return=(fact-iter 3 1)]
[n=3,return=?],[n=3,acc=1,return=(fact-iter 2 3)]
[n=3,return=?],[n=3,acc=1,return=?],[n=2,acc=3,return=(fact-iter 1 6)]
[n=3,return=?],[n=3,acc=1,return=?],[n=2,acc=3,return=?],[n=1,acc=6,return=6]
[n=3,return=?],[n=3,acc=1,return=?],[n=2,acc=3,return=6]
[n=3,return=?],[n=3,acc=1,return=6]
[n=3,return=6]
```

Tail position and tail call

> The ***tail position*** of a sequence of expressions is the last expression of that sequence.

```
(lambda ()
  exp1
  ; ...
  expn) <-- tail position
```

> When a function call is in the tail position we named it the ***tail call***.

```
(lambda ()
  exp1
  ; ...
  (f ...)) <-- f is a tail call
```

Tail call and the call stack

A tail call does not need to push a new activation frame! Instead, the called function can "reuse" the frame of the current function. For instance, in `(fact 3)`, the call `(fact-iter 3 1)` is a tail call.

```
[n=3,return=(fact-iter 3 1)]
[n=3,return=?],[n=3,acc=1,return=(fact-iter 2 3)]
```

Can be rewritten with:

```
[n=3,return=(fact-iter 3 1)]
[n=3,acc=1,return=(fact-iter 2 3)]
```

In attempt #2, both calls to `fact-iter` are tail calls.

Tail-Call Optimization

- Eschews the need to allocate a new activation frame
- In a recursive tail call, the compiler can convert the recursive call into a loop, which is more efficient to run (recall our $5\times$ speedup)

The tail-recursive optimization pattern

Tail-recursive `map`, using the generalized tail-recursion optimization pattern

```
(define (map f l)
  (define (map-iter accum l)
    (match l
      [(list) (accum (list))]
      [(list h l ...) (map-iter (lambda (x) (accum (cons (f h) x))) l)]))
  (map-iter (lambda (x) x) l))
```

The accumulator delays the application of `(cons (f (first l)) ?)`.

1. The initial accumulator is `(lambda (x) x)`, which simply returns whatever list is passed to it.
2. The base case triggers the computation of the accumulator, by passing it an empty list.
3. In the inductive case, we just augment the accumulator to take a list `x`, and return `(cons (f h) x)` to the next accumulator.

> The accumulator works like a pipeline: each inductive step adds a new stage to the pipeline, and the base case runs the pipeline: `(stage3 (stage2 (stage1 ((lambda (x) x) nil))))`

Tail-recursive `map` run

```
(map f (list 1 2 3)) =
; First, build the pipeline accumulator
(define (accum0 x) x) (map-iter accum0 (list 1 2 3)) =
(define (accum1 x) (accum0 (cons (f 1) x))) (map-iter accum1 (list 2 3)) =
(define (accum2 x) (accum1 (cons (f 2) x))) (map-iter accum2 (list 3)) =
(define (accum3 x) (accum2 (cons (f 3) x))) (map-iter accum3 (list)) =
; Second, run the pipeline accumulator
(accum3 (list)) =
(accum2 (list (f 3))) =
(accum1 (list (f 2) (f 3))) =
(accum0 (list (f 1) (f 2) (f 3))) =
(list (f 1) (f 2) (f 3)))
```