Quick Scheme

AUBURN UNIVERSITY

Rodrigo Sardiñas

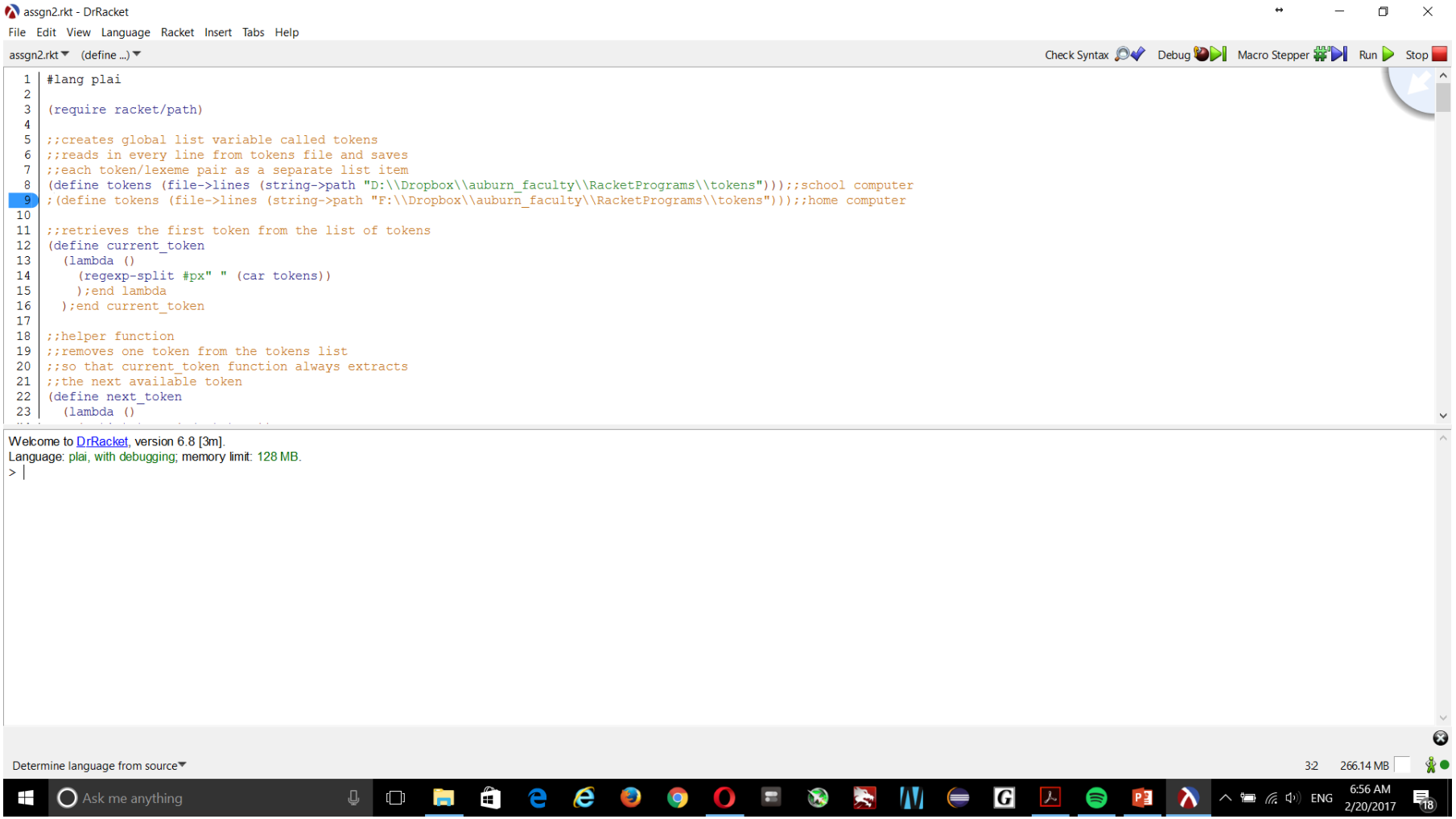
Topics

AUBURN

- Dr. Racket
  - Interactive Mode
- Lambda Calculus (short)
- Lists!
- Variables
- Procedures (lambda)
- Selection (cond)
- Iteration (recursive)

Dr. Racket (Gui Demo In-class)

AUBURN



Dr. Racket

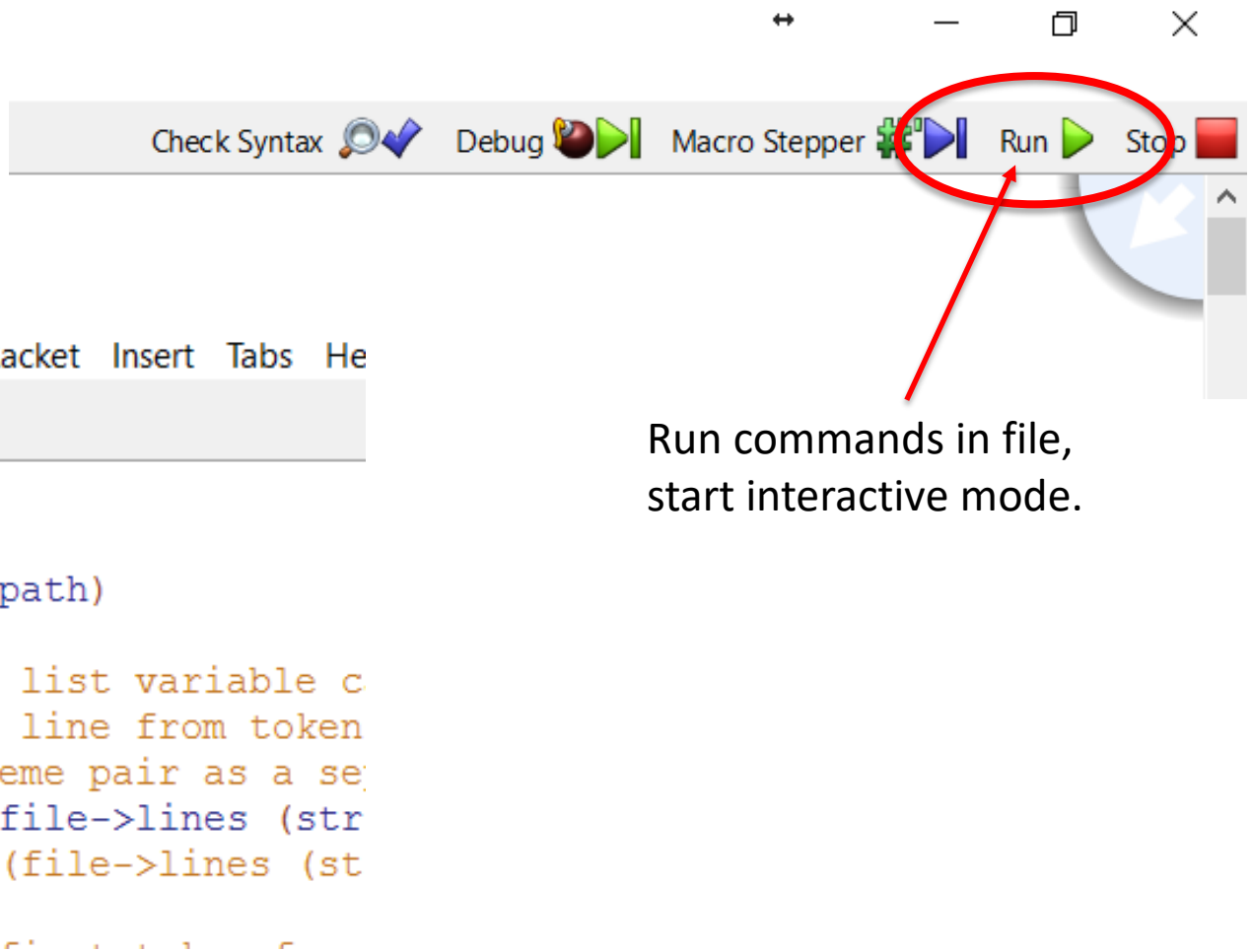
AUBURN

- Where to get it
  - https://racket-lang.org
- Reference Docs
  - http://docs.racket-lang.org/drracket/index.html
  - http://docs.racket-lang.org/quick/index.html
  - https://docs.racket-lang.org/racket-cheat/index.html
- Online Textbook
  - http://www.scheme.com/tspl4/

Dr. Racket – Interactive & Language

AUBURN

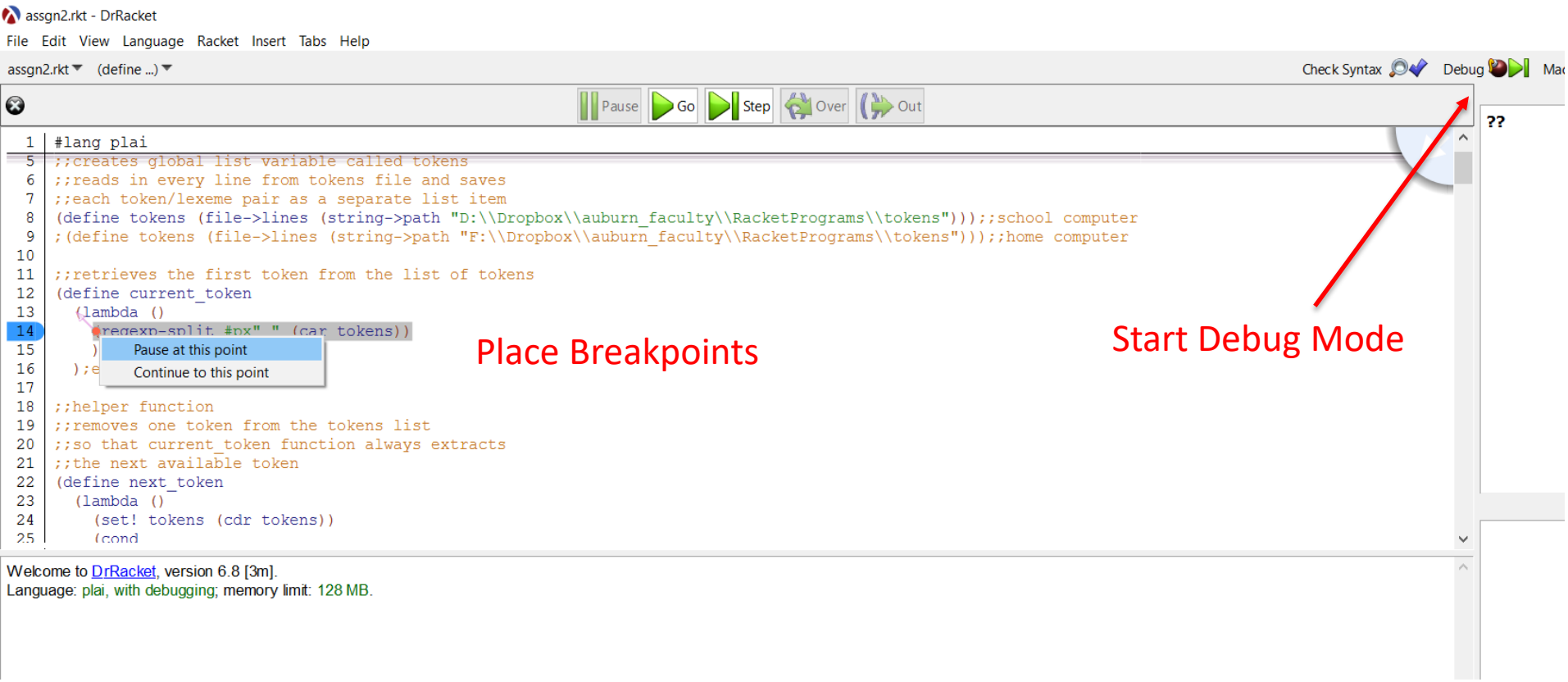
Use this “language”. Kind of like a library.



Run commands in file, start interactive mode.

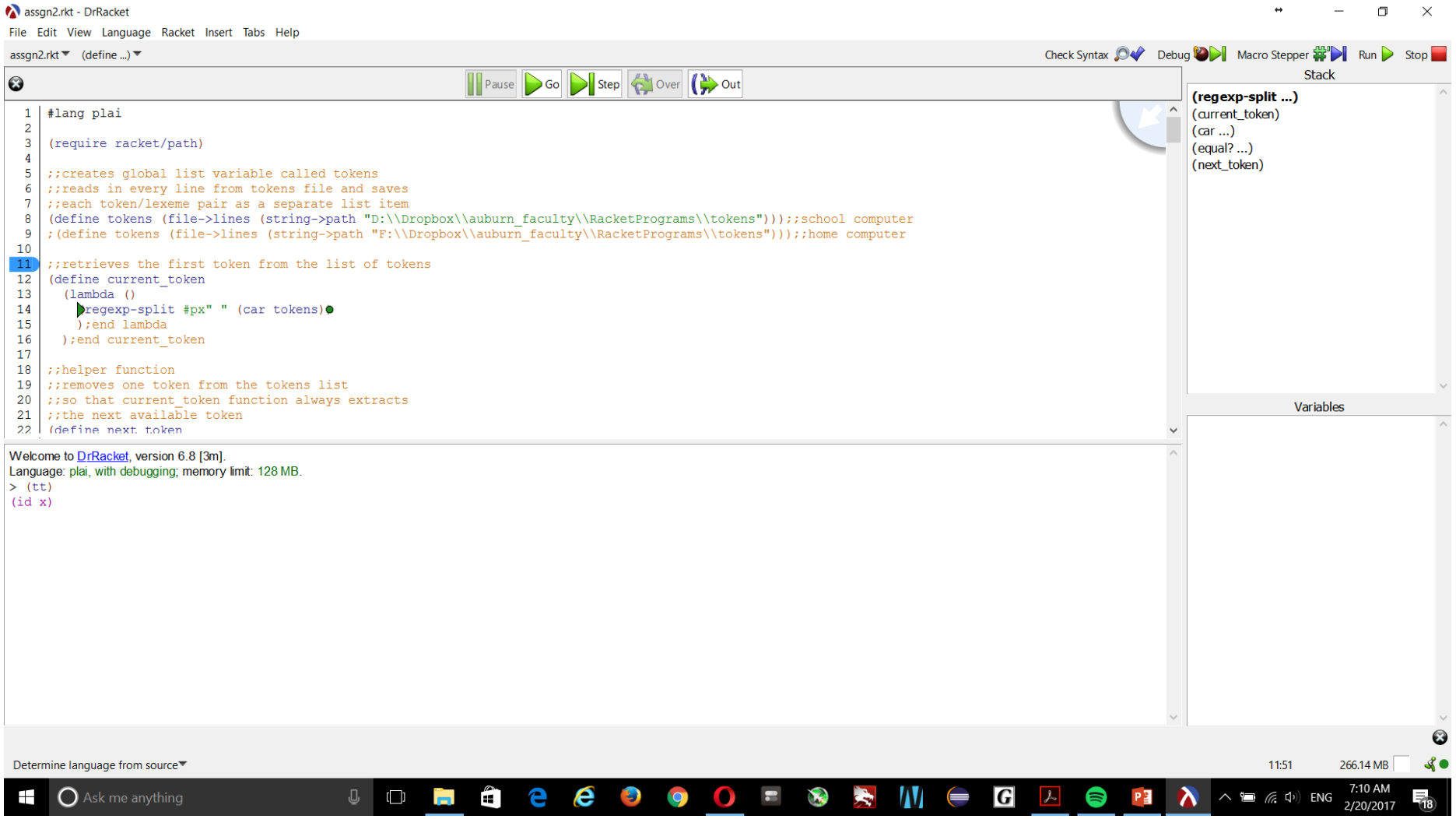
Dr. Racket - Debugging

AUBURN



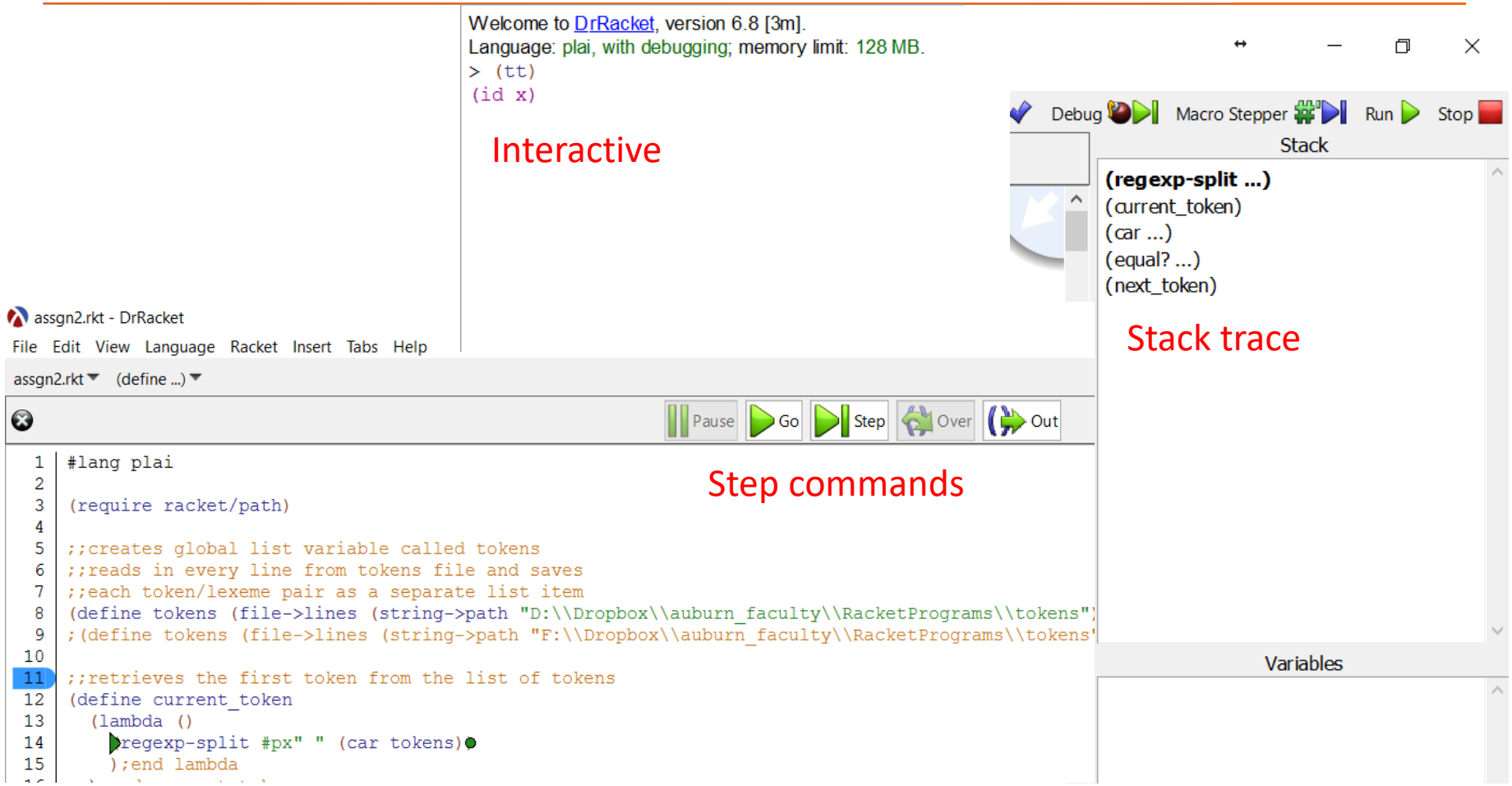
Dr. Racket - Debugging

AUBURN



Dr. Racket - Debugging

AUBURN



Lambda Calculus

AUBURN

Lambda calculus (also written as λ-calculus or called "the lambda calculus") is a formal system in mathematical logic and computer science for expressing computation by way of variable binding and substitution.

Wikipedia

Lambda Calculus

AUBURN

- Short, but interesting paper
- Perhaps more simply put here
- Alonzo Church
- Smallest Universal Programming Language!
- No Data types, only functions

Lists! It's a function! No, it's a variable! No… It's a '(List)!

AUBURN

- (car (first element in list))
  - (car '(A B C)) – returns '(A)
- (cdr (everything except first item in list – the rest))
  - (cdr '(A B C)) – returns '(B C)
- (cadr & caddr): shorthand combinations of the above
- (first (first element in list; list with 1 item))
- (second … tenth (nth item in list; list with 1 item))
- (cons (combine two "lists"))
  - (cons 'A '(B C D)) – returns '(A B C D)
- list (similar to cons, creates new lists containing elements)
  - (list 'A 'B '(C D)) – returns '(A B (C D))

Lists

AUBURN

- Given the way that scheme is implemented EVERYTHING returns something, even lists!
- If a list only accepts 2 items, and you need to pass in multiple items, the 2nd item must be another list
- Lists can be heterogeneous in Scheme (this extends to data & functions)
- Scheme uses pre-fix notation. So the first list item is always the “root” of an expansion

Variables - global

AUBURN

- Even variables are defined as lists in scheme
- Global (using define)
  - (define x 1) creates a global variable called x that is initialized to 1
    - First list item: define
    - Second list item: x
    - Third list item: some value, substitute this value for x
  - (define x (+ 1 2)) creates a global variable called x that is initialized to 3

Variables – global (list items)

AUBURN

- Even variables are defined as lists in scheme
  - (define x 1)
    list item 1 list item 2 list item 3
  - (define x (+ 1 2))
    list item 1 list item 2 list item 3 (the whole list)

sub li 1 sub li 2 sub li 3

Variables - local

AUBURN

- Global (using let)
  - (let

    (x (+ 1 2))

    (y (+ 2 2))

    x + y) creates local variables x and y initialized to 3 and 4
- By default, procedures (let is a procedure) always return the value of the last item in the list (x + y)
- See link above for an example of returning last item

Variables - modfying

AUBURN

- Using set!
- Variables or definitions can not be “redefined” once initialized. To change the value of a definition, use the set! Keyword.

(define x 0)

(set! x 10)

Procedures (lambda)

AUBURN

- Procedures (functions) are also described as lists
- Using lambda

(lambda (parameters go here)

Function body goes here (as multiple lists of instructions) ***… see sample code on canvas***

)

- Used this way, lambda is a “one-time” anonymous function

Procedures (lambda)

AUBURN

- Assigning a name to a lambda
- Using lambda

(define x

(lambda (parameters go here, can be empty)

Function body goes here (as multiple lists of instructions) ***… see sample code on canvas***

))

- Calling x will now execute the lambda procedure associated with x

Procedure (define)

AUBURN

- You can also “define” procedures
- Symbols can represent data, or procedures
- (x) is a procedure (a list with 1 item in it that is a procedure)
- x or ‘(x) is data (a list with 1 item in it that is data – how we traditionally view a variable)

Procedure (define)

AUBURN

```
(define x
    (display “procedure?”)
) ; will not display procedure, returns void

(define (x) ← shorthand for procedure
    (display “procedure?”) ← body is rest of list items
); returns void, displays “procedure?”
```

Procedures

AUBURN

```
(define (x)
    (display “procedure shorthand”))

(define x
    (lambda ()
        (display “Previous example converted to this”) ) )
```

Procedures

AUBURN

```
(define (x a)
    (+ a a)
    (display “Shorthand for below”))

(define x
    (lambda (a)
        (+a a)
        (display “What really happens above”) ) )
```

Procedures (as lists)

AUBURN

1 2 1 2 3 1 2

(define (x a) (+ a a) (display “Shorthand for below”))

1 2 3 4

Lambda: 1 2 3 4

w/in lambda: 1 2 3 1 2

(define x (lambda (a) (+a a) (display “Actual”)))

1 2 3

Procedures (operators as procedures)

AUBURN

```
(define (= x y)
     (if (equal? x y) #t #f)
      )

(define =
    (lambda (x y)
        (if (equal? x y) #t #f)
     ) )
```

Selection (cond)

AUBURN

- Multiple ways, we will cover cond
- Several benefits
  - Inherently handles “blocks” with multiple commands
    - With if statements you’d have to code a begin block
    - Begin blocks are implicit with the cond statement
- Example on next slides

Selection (cond)

AUBURN

```
(cond
    (
        (= x 10)
        (what to do if true?)
        (anything else)
        (more instructions?)
    ) ;end first equal block can have more
);end cond
```

Selection (cond)

AUBURN

```
(cond                (with else)
    ((= x 10)
        (do some stuff)) ;end first equal block
    (else
        (body of else goes here. Multiple ok))
);end cond
```

Selection (cond) – equivalent, required for *if* statement

AUBURN

```
(cond
    ((= x 10)
        (what to do if true?)
        (anything else)
    ) ;end first equal
);end cond
```

```
(cond
    ((= x 10)
        (begin
            (what to do if true?)
            (anything else)
        ); end begin block
    ) ;end first equal
);end cond
```

Iteration – write it recursively!

AUBURN

- There are for and do loops in scheme, but ideally we want to write recursive functions
- Different way to think about coding
- Specifically, we want to write tail-recursive functions
- Very fast in scheme
- Scheme guarantees that tail recursion will be converted to iterative when executed/compiled
- See *next_token* in homework sample