CSCC24 – Principles of Programming Languages
Formal Language Theory

Anya Tafliovich[1]

[1] with thanks to S.McIlraigh, G.Penn, P.Ragde

Scheme/Racket – jumping right in

- A Functional Programming Language
    - actually...
      an imperative programming language
    - with a functional core
- static scoping
- dynamic typing
- clear and simple syntax
- clear and simple semantics (for the core): λ-calculus
- uniform treatment of program and data
- implementations properly tail-recursive
- functions are values: created dynamically, stored, passed as parameters, returned as results, etc.
- pass-by-value

expressions

- Any value is an expression.

```
<expr> ::= <val>
```

- The application of a function to some number of argument expressions is an expression.

```
<expr> ::= ( <func> <expr> ... )
```

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

datatypes

- **number**: `5`, `-7`, `3.14`, …
  Includes integers, rationals, reals, and complex numbers.
  These types are not disjoint, they form a hierarchy.

  Some useful procedures:
  - type predicates: `number?`, `integer?`, `real?`, `complex?`
  - other predicates: `positive?`, `negative?`, `zero?`, `even?`, `odd?`, …
  - arithmetic operations: `+`, `-`, `*`, `/`, `floor`, `remainder`, …
  - comparison predicates: `=`, `<`, `<=`, `>`, `>=`

datatypes

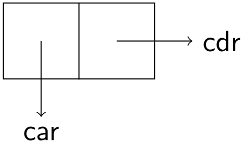
- **symbol**: `'a`, `'ABC`, `'foo`, ...

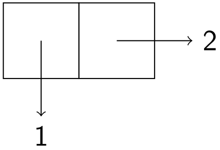
  Also
  `'with-a-dash`, `'questions?-too?`, `'2strange4use`, ...

- **boolean**: {`#t, #f`}.
  - Useful synonyms: `true, false`.
  - Type predicate: `boolean?`.
  - Useful procedures: `not`, `boolean=?`.

datatypes

- **pair**: `'(1 . 2)`, `'(1.12 . -4)`, `'(1 . a)`, ...

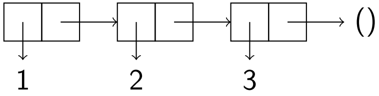




  - Constructor : `cons`.
    - `(cons 1 2)`
    - `(cons (cons 1 2) 3)`
  - Selectors : `car`, `cdr`.
    - `(car '(1 . 2))`
    - `(cdr '((1 . 2) . 3))`
    - `(car 1)`
    - `(cdr '())`

datatypes

- **list**: `'()`, `'(a b)`, `'(1 2 foo 3.14)`, ...
  - The only list that is not a pair is `'()`.
  - If `L` is a list and `V` is a value, then `(cons V L)` is a list.
    - `(cons 1 '())`
    - `(cons '(1) '(2 3))`
    - `(car '(1))`
    - `(cdr '(1))`
    - `(cdr '(1 2 3))`



- Better selector names for lists: `first`, `rest`.

datatypes

- **strings**: `"hello, world!"`
- **vectors**: `#("a" "b" "c")`
- **hash tables**:

  ```
  #hash(("apple" . red)
        ("banana" . yellow))
  ```
- and more

but we won't use them much in this course.

values and evaluation

- number, boolean, etc.: it's own value

- `(quote expr)` : `expr`
  `'expr` : shortcut for `(quote expr)`

- lists:
  - `(f . (arg0 ... argN))`
    has value
    $f(arg0, ..., argN)$

  This is called prefix notation.

Read-Eval-Print Loop

Read: Read input from user.

Eval: Evaluate input.
Lists are evaluated as follows:
`(f` $\texttt{arg}_0$ $\texttt{arg}_1$ `...`$\texttt{arg}_n$`)`

1. Evaluate `f` (a procedure) and each $\texttt{arg}_i$:
   - in general: in an unspecified order!
   - in PLT Scheme + Racket: left to right
2. Apply procedure to argument values.

Print: Print resulting value:
the result of the procedure application.

Read-Eval-Print Loop example

```
(cons 'a (cons 'b '(c d)))  =>  '(a b c d)
```

1. Read the input `(cons 'a (cons 'b '(c d)))`.
2. Evaluate `cons` : get the cons procedure.
3. Evaluate `'a` : get symbol `'a` .
4. Evaluate `(cons 'b '(c d))` :
   4.1 Evaluate `cons` : get the cons procedure.
   4.2 Evaluate `'b` : get symbol `'b` .
   4.3 Evaluate `'(c d)` : get list `'(c d)` .
   4.4 Apply the `cons` procedure to `'b` and `'(c d)` : get list `'(b c d)`.
5. Apply the `cons` procedure to `'a` and `'(b c d)` : get list `'(a b c d)`.
6. Print the result of the application: `'(a b c d)`.

evaluation by substitution

```
(+ (* 3 4) (- 6 5))
=> (+ 12 (- 6 5))
=> (+ 12 1)
=> 13
```

Substitution model: An expression is reduced to a value by repeatedly finding the leftmost expression ready for substitution (all arguments are values) and replacing it with its value.

The substitution model provides good intuition for the pure subset of Scheme.

values and evaluation

- lists:
  - `(f . (arg0 ... arg N))`
    has value
    $f(arg0, ..., argN)$

Not all lists have values:

```
(1 2 3) => ?
```

special forms

special forms: (**syntactic-keyword** `expr`)

Some syntactic keywords: `and`, `or`, `if`, `cond`.

```
<expr> ::= (and { <expr> } )
<expr> ::= (or { <expr> } )
```

special forms — if

Syntax:

```
<expr> ::= (if <expr> <expr> <expr>)
```

Semantics:

```
(if condition expr0 expr1)
```

1. Evaluate `condition`.
2. If result is not `#f`, evaluate `expr0`.
3. Otherwise, evaluate `expr1`.

```
> (if false (car '()) 42)
42
```

special forms — cond

Syntax:

```
<expr> ::= (cond {(<expr> <expr>)} [(else <expr>)])
```

Semantics:

```
(cond [(cond0  expr0)
       (cond1  expr1)
         ....
       (else   exprN)])
```

- Evaluate `cond0`.
- If the result is not `#f`, evaluate `expr0`.
- Otherwise, evaluate `cond1`.
- If the result is not `#f`, ...

More syntactic keywords: `define`, `lambda`.

definition

```
<defn> ::= (define <id> <expr>)
```

A definition binds a name to a value.

```
(define x 7)          ; binds x to 7
(define y (* x x))    ; binds y to 49
```

A subsequent use of the name results in a substitution of the associated value.

lambda expression

```
<lambda> ::= (lambda (<arg> ...) <expr> ...)
```

Function with zero or more arguments `<arg> ...` and body `<expr> ...`

```
> (lambda (x y) (+ x y))
#<procedure>
> ((lambda (x y) (+ x y))
   1 2)
3
```

More on semantics of function application later.

evaluation

Could we in, say, Python, define a function called `my_and` so that the call `my_and(a,b)` behaves exactly like Python's expression

```
a and b
```

I. e., if `a` evaluates to false, then `b` is not evaluated?

What about `my_if(cond, a, b)` that behaves exacly like

```
if cond:
    a
else:
    b
```

indentation

Write code so that it is readable by other human beings.

```
> (define max
    (lambda (x y)
      (if (> x y) x y)))
> (max 3 4)
```

Use a good editor: it will take care of indentation for you.

higher-order procedures

Procedures as input values:

```
; (all-ok? ok? lst) -> boolean?
; ok? : procedure applicable to every element of lst
; lst : list?
(define (all-ok? ok? lst)
```

higher-order procedures

Procedures as returned values:

```
; (make-proc f g) -> procedure?
; f,g : procedure, the composition f o g is well-defined
; return a composition f o g ( aka f(g(x)) )
```

higher-order procedures: `map`

Let's define our own restricted version first.

- `mymap` takes two arguments: a function and a list.
- `mymap` builds a new list whose elements are the result of applying the function to each element of the (old) list.
- The built-in `map` is more general and more powerful.

higher-order procedures: `map`

`(map proc l1 l2 ...  ln)`

- `proc` : an n-ary procedure
- `l1 l2 ...  ln` : lists of length `m`
- apply `proc` element-wise to elements of
  `l1 l2 ...  ln`
- return the list of results `(e1 e2 ...  em)`
- in general, the order of evaluation is unspecified: appears to be evaluated in parallel
- specific implementation may define the order as left to right

higher-order procedures: map

```
(map proc l1 l2 ...  ln)

(map abs '(-1 2 -3 4 5)) ==>

(map max '(1 2 3) '(0 5 42)) ==>

(map + '(1 2 3) '(2 3 4) '(3 4 5)) ==>

(map cons '(a b c) '((a a) (b b) (c c))) ==>

(map (lambda (x) (+ x 1)) '(1 2 3)) ==>

(map (lambda (x y) (+ 1 (- y x)))
     '(0 5 10)
     '(1 5 11))
==>
```

higher-order procedures: `fold`

`(foldr op id lst)`

- op : a binary procedure
- `lst` : list of arguments
- apply op right-associatively to elements of `lst`
- return result of evaluation
- the identity element `id` is always used

That is:

- `(foldr op id '())` ⇒ `id`
- `(foldr op id '(e))` ⇒ `(op e id)`
- `(foldr op id '(e1 e2 ... en))`
  ⇒ `(op e1 (op e2 (op ... (op en id))))`

higher-order procedures: `foldr`

Let's define our version first.

higher-order procedures: `foldr`

`(foldr + 0 '(1 2 3 4))` $\Rightarrow$

higher-order procedures: `foldr`

`(foldr list '() '(1 2 3 4))` ⇒

higher-order procedures: `foldr`

`(foldr op id lst)`

- `op` ⇔ `cons`
- `id` ⇔ `'()`

```
   (1 2 3 4)
=> (cons 1 (cons 2 (cons 3 (cons 4 '()))))

   ( +   1 ( +   2 ( +   3 ( +   4 0 ))))
=> (foldr + 0 '(1 2 3 4)

   (list 1 (list 2 (list 3 (list 4 '()))))
=> (foldr list '() '(1 2 3 4)
```

higher-order procedures: `foldl`

`(foldl op id lst)`

- op : an binary procedure
- `lst` : list of arguments
- apply op left-associatively to elements of `lst`
- return result of evaluation
- the identity element `id` is always used

That is:

- `(foldl op id '())` ⇒ `id`
- `(foldl op id '(e))` ⇒ `(op e id)`
- `(foldl op id '(e1 e2 ... en))`
  ⇒ `(op en (op en-1 (op ... (op e1 id))))`

higher-order procedures: `foldl`

`(foldl op id lst)`

In other implementations:

- `(foldl op id '())` ⇒ `id`
- `(foldl op id '(e))` ⇒ `(op id e)`
- `(foldl op id '(e1 e2 ... en))`
  ⇒ `(op ... (op (op id e1) e2) ...) en )`

higher-order procedures: `apply`

```
(apply op lst)
```

- `op` : an *n*-ary procedure
- `lst` : list of *n* arguments to `op`
- apply `op` to elements of `lst`
- return result of evaluation

higher-order procedures: `apply`

`(apply op lst)`

- `(apply + '(1 2 3 4))`
  $\Rightarrow$
- `(apply cons '(a (b c)))`
  $\Rightarrow$
- `(apply list '(1 2 3 4))`
  $\Rightarrow$
- `(apply cons '(a (b c) (d)))`
  $\Rightarrow$

higher-order procedures: `foldr`

`(foldr op id lst)`

- `op` ⇔ `cons`
- `id` ⇔ `'()`

```
(define (append list1 list2)
```

```
(define (map p lst)
```

higher-order procedures: foldr

```
   (append '(1 2 3 4) '(5 6 7))
=>
```