CSI2120 Programming Paradigms

Jochen Lang

jlang@uottawa.ca

Functional Programming in Lisp

- Language design[illegible]n McCarthy between 1956 - 1959 at MIT for [illegible]ns related to artificial intelligence
  - one of the oldest languages still in use
- LISP = LISt Processor.
- Derived from λ-calculus.
  - λ-calculus allows functions to be the values of an expression
- Many dialects
  - Lisp 1.5 (1960), Scheme (1975), Common Lisp (1985)…
- Rich language: functional, symbolic.
- Syntax and semantics are simple and uniform

Creation of Lisp

- 1960: McCarthy [illegible] his paper on Lisp
- Lisp/Scheme ha[illegible]mple operators and a rich notation for functions.
  - This is combined with simple data structures
- As a result we have a full and expressive programming language

Nine Key Concepts

1. Conditions (if-t
2. Functions as da
3. Recursions
4. Variables as pointers
5. Automatic garbage collection
6. A program is an expression (not a series of statements)
7. Symbols or atoms
8. Lists and trees
9. Complete language available at all times (read-eval-print)

Pure Functional Programming

- A program corresponds to a function call
- A function is a composition of functions
- Functions are non-causal
  - Depend only on the parameters passed
- No variables, no assignments
- No loops, no control statement
  - Beyond the if-then-else function

Functional Programming in Practise

- Some additions to pure functional programming
  - Local definitions, even of plain values
  - Assignments (lexically scoped variables)
  - Use an execution sequence (in order to break up the program).

Functional Programming in Scheme

- Scheme is LISP [illegible]signed at MIT in 1975, mainly for education
- Initially small
  - But it is now a complete language.
- Standardized by ANSI / IEEE
  - Language continues to evolve
- Commonly used as interpreted language
  - But may also be compiled to be executed efficiently.

Application Area of Functional Programmin

- Applications of sy mputation, i.e., non-numerical applications
  - Artificial intell pert systems, natural language interfaces, ...)
  - Automated reasoning (theorem proving, proofs of programs …)
  - Symbolic Computation
  - Games
- Today, functional programming is everywhere
  - Python lambda, map, filter, reduce, list comprehension etc.
  - Javascript function expressions, binding, currying, partials
  - Also C++ and Go have lambdas, higher order functions etc.
  - Many other languages …

Basic Concepts

- The list is the fu[illegible]l data structure
- Atom: a number [illegible]or a symbol.
  - All data types are equal
- An expression is either an atom or a list
- A List is a series of expressions in parenthesis
  - Including the empty list ()
- A function is a first class object (first-class data) that can be created, assigned to variables, passed as a parameter or returned as a value.

Evaluations of Expressions

- Constants are ev what they are.
  - numbers

```
> 2    => 2
```

  - strings

```
> "Hello"  => "Hello"
```

- Identifiers evaluate to the value that is attributed to them.

```
> *  => #<procedure:*>
```

- Lists are evaluated by
  - first evaluating the head, i.e., the first expression; the value of this expression must be a function
  - The arguments of this function are the values obtained by evaluating the expressions contained in the rest of the list

A First Scheme Session

- In its simplest form, a Scheme interpreter session uses the interactive READ-EVAL-PRINT programming model (REPL)
- Example:

```
> (+ 3 4)
7
```


- In the example, we have a list.
  - The first entry is the function +
  - The rest of the list are constant expressions 3 and 4.
- The list is read as `(+ 3 4)`, evaluated and then the result `7` printed.

Scheme Interpreter

- Classical MIT Sc rpreter
  - http://www.g ftware/mit-scheme/
  - While available, not well supported under windows
- Racket
  - Another LISP dialect
- DrRacket
  - http://racket-lang.org/
  - Convenient and full-fledged programming environment to run LISP dialects

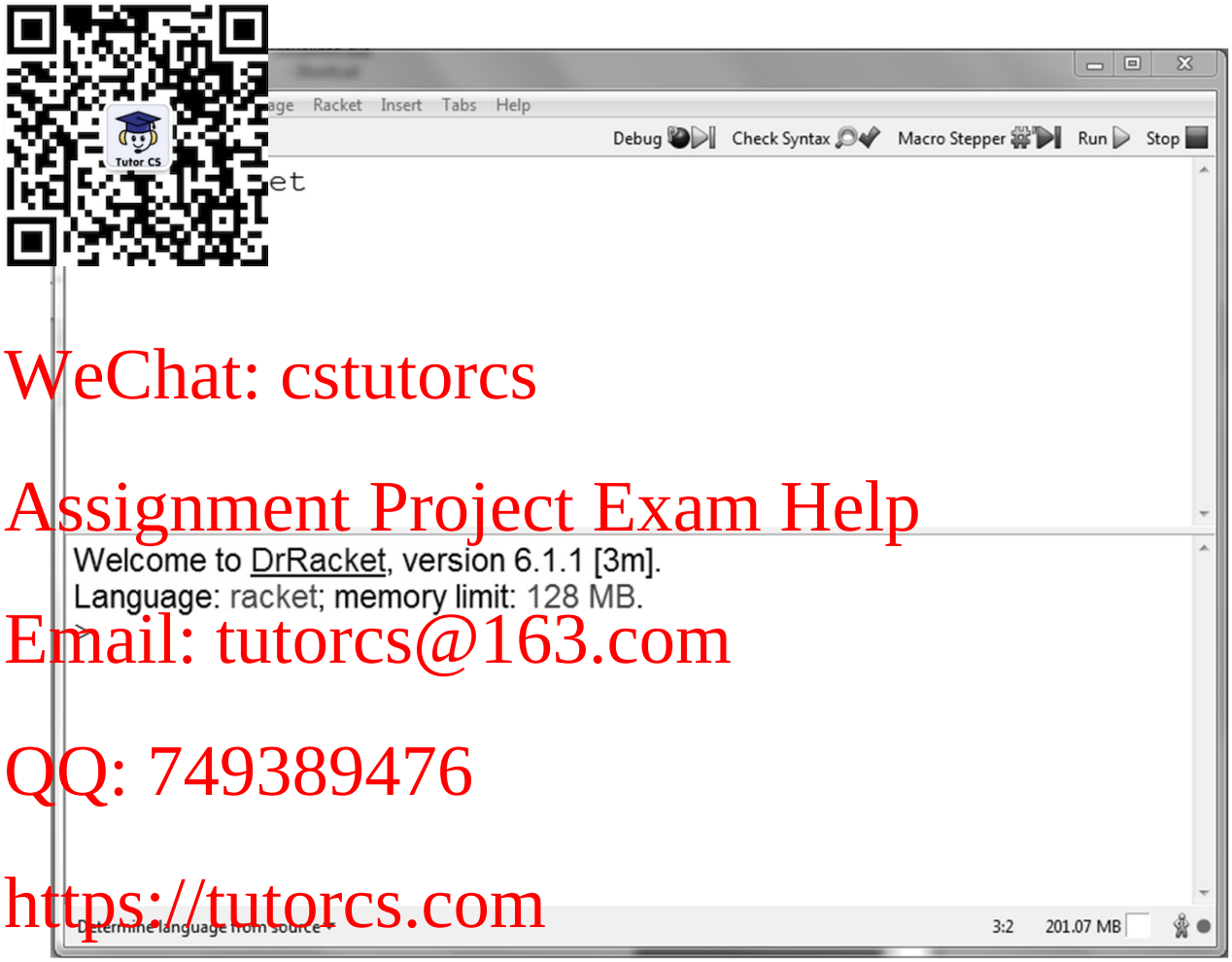
Few Remarks on the IDE

You must select a language in top of window.

Program editor

Bottom window is for running programs

This is the classic REPL Buffer



Jochen Lang, EECS
jlang@eecs.uOttawa.ca

Evaluation of Expressions

- The prefix notation is used in expressions
  - Instead of infix operators as in 3+4*5
  - One needs to write (+ 3 (* 4 5))
- To evaluate an expression, all sub-expressions must be evaluated first.
  - The evaluation follows normal order evaluation
  - Brackets determine the order and are not optional

```
(+ 3 (* 4 5))
(+ 3 20)
23
```

Special Syntactic Forms

- Some expressions do not obey the normal order evaluation, these expressions are said to have a *special syntactic form*.
- The evaluation of their arguments is deferred until the required values are known.
- The main special syntactic forms are:
  - if statement
    - conditional branching
  - creation of local scope
  - quotation

Special Syntactic Forms: The Alternative (if statement)

```
(if (= x 0) in 1 x))
```

- The expression f he if is evaluated first .
- If its value is true (# t) then
  - the second argument is evaluated
  - its value is returned without evaluating the third argument
- if its value is false (# f) then
  - the third argument is evaluated and returned

Special Syntactic Forms: 2. Conditional Branching

- Conditional expressions are similar to `if` but more than two branches can be specified

```
(cond ((< x xmin) xmin) ((> x xmax) xmax) (#t x))
```

- The `cond` expression is followed by a series of lists composed of two expressions.
  - If the first of the two expressions in one of these lists evaluates to #t then the value of the second expression is returned
  - If the first expression evaluates to false, then evaluation proceeds to the next list.
  - If no lists evaluates to #t then nil is returned.

Example: Conditional and Top-level Define

Definition of the fu wIt taking an argument `pts` and evaluating a c

```
(define (showIt pts)
  (cond ((or (<= pts 3) (>= pts 65)) 0)
        ((<= 4 pts 6) 0.5)
        ((<= 7 pts 12) 1.0)
        ((<= 13 pts 15) 1.5)
        ((<= 16 pts 18) 1.8)
        (else 2.0)))
=> showIt
(showIt 25)
=> 2.0
```

Special Syntactic Forms: 3. Creating Local Scope

- Let Expressions

```
(let ((a 3) (b 4)) (* a b))
=> 12
```

  - The first argument of a let expression is a list of links created between an identifier and a value
  - These links are only valid for the evaluation of the following expression(s)
    - There can be several to allow the execution of a sequence.

```
(let ((a 3) (b 4)) (* a b) (+ a b))
=> 12 7
```

Special Syntactic Forms: 4. Quotation

- The quote function specifies that an argument list is not evaluated.

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

  - But the list is rather returned as is.
  - Quotation is necessary here, otherwise the first expression of a list needs to evaluate to a function.

```
'(+ 3 4) => (+ 3 4)
(+ 3 4) => 7
```

  - The quote function can simply written with a ' :

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

Quotation Example

```
(let ((a '(1 2 3)) (b '(3 4 5))) (cons a b))
```

equates to

```
(cons '(1 2 3) '(3 4 5))
```

⇒ `((1 2 3) 3 4 5)`

- The function cons is the dot operator for lists, i.e., it puts the first expression as the head of the second list expression
- The list `( 1 2 3 )` becomes the first element in the combined list `((1 2 3) 3 4 5)`
- (Much) more on list processing soon.

Example: Building a List with the Function `lis`

```
(list `a `b `c)
⇒ (a b c)
```

```
(list `(a b c))
⇒ ((a b c))
```

Lambdas

- Lambda expressions: "anonymous local functions"
  The expression `(lambda (var1, var2, …) exp1 exp2 …)` returns a function and applies it to the variables that are the parameters to the function expressions, e.g.,

```
((lambda (x) (* x x)) 3)
⇒ 9
```

  Multiple variables and multiple expression (result of last expression is the answer

```
((lambda (f x y) (f x x) (f x y) (f y y)) +
2 3 )
⇒ 6
```

Function Definitions

A definition associates a value (or expression) to a name:

```
(define square (lambda (x) (* x x)))
```

or equivalently (and shorter):

```
(define (square x) (* x x))
```

Use of define, here procedure square:

```
(square 2)
⇒ 4
```

Example: Factorial

Top-level Function

```
(define (fact n)
  ( if (> n 0)
       ( * n (fact (- n 1)))
       1)
  )
=> fact

(fact 35)
=> 10333147966386144929666651337523200000000
```

Function Definitions with Lambdas

- We have seen La be combined with top-level defines

```
(define fct (lamb (f x x)))
(fct + 13)
=> 26
(fct * 4)
=> 16
```

- Combine with let binding: x is a let-bound variable in the enclosing scope

```
(let ((x 'a))
  (let ((f (lambda (y) (list x y))))
    (f 'b)))
=> (a b)
```

Lambda Expression and Let-Bound Variables


```
(let ((x 2) (y 3)) (+ x y))
```

is equivalent to


```
((lambda (x y) (+ x y)) 2 3)
```


In general:


```
((let (var val) …) expr…) ≡ ((lambda (var …)
  expr…) val…)
```


Jochen Lang, EECS
jlang@uOttawa.ca

Example: Greatest Common Divisor (GCD)

- top level define function

```
(define gcd
  (lambda (a b)
    (if (= a b)
      a
      (if (> a b)
          (gcd (- a b) b)
          (gcd a (- b a))))))
=> gcd
(gcd 12 15)
=> 3
```

Summary

- Introduction to Functional Programming
- Basic Scheme
- Special Syntactic Forms
  - Alternative (if then else)
  - Conditional
  - Local Scope
  - Quotation
- Let-bound variables
- Top-level (function) definitions
- Lambda expressions

Jochen Lang, EECS
jlang@uOttawa.ca