# Autotest Generator CS135

By: Sam (Tangrui Song)
ID: trsong
Term: Fall 2013

## Contents

# 0 Introduction

Autotest Generator (testgen.ss) is a racket script which is used to generate all autotest cases simultaneously. It is a really powerful tool which was designed for saving your time and effort.

Before you start reading this document, make sure you have already read "AutotestCreationFall2011" by YanTingChen. Because the implementation of Autotest Generator was totally based on "AutotestCreationFall2011".
This document was created on Dec 6, 2013 (Fall2013) by Sam. It may or may not be applicable for future term. The following document may use Autotest Generator and testgen interchangeably. They are exactly the same concept.

Chapter 1 to 4 give you basic knowledge of how to use Autotest Generator and how it works. However, if you are good at learning through examples, it is a good idea to go to Chapter 5 Application first. Chapter 1 tells you everything you should know before making tests. Chapter 2 introduces a list of useful helper functions provided. Chapter 3 teaches how to initialize and generate tests in details. Chapter 4 provides the usage of different generating modes. Chapter 5 is the real time application of the Autotest Generator.

# 1 Preparations

Before you write any kind of tests, it is really important to **understand** the questions. All test case must be written under the restrictions. Here is a list of **steps** which you need to follow:

(a) Do assignments.
(b) Watch restrictions.
(c) Get **sample solutions** from instructors/ other ISAs.
(d) Put sample solutions into one single file (eg. Assn.rkt)
(e) Swith lang level in Assn.rkt to **lang-racket**
eg. add the line ***#lang racket*** at the top of Assn.rkt and change language to Determine Language from Source
(f) Put **testgen.ss** into the same directory
(g) Add the line ***(require "testgen.ss")*** after #lang racket
**warning:** do **NOT** copy directly from this document

You can also find the procedure in the slides of "**CS135 Autotest Generator.pdf**".

## 1.1 Create Test Cases

***Test:*** is a list of inputs Autotest Generator can use.
Suppose there is a function called zero?.
(I)

```
;; zero? : Number -> Boolean
```

(define test4zero? (list -1 0 1))

Here we have created 3 test cases: (zero? -1)  (zero? 0)  (zero? 1)

(II)

What about list function.

;; sum: (listof Nat) -> Nat

;; sum consume a list of Nat and produce the sum of that list.

```
(define test4sum (list  empty
                        (list 1)
                        (list -1 1 2) ))
```

It creates 3 test cases. (sum empty)  (sum (list 1)) (sum (list -1 1 2)).

(III)

What about a function which has more than one parameters.

;; my-max: Num Num Num -> Num

;; my-max consumes three numbers and produces the maximum one.

```
(define test4my-max (list (list 0 0 0)
                          (list 1 2 3)
                          (list -1 1 2)))
```

Test: (test4my-max 0 0 0)

(test4my-max 1 2 3)

(test4my-max -1 1 2)

If you look at (II)'s test case you will find it looks smiliar, and we will talk about how Autotest Generator could handle (II) and (III) in chapter 4 Test Generating Mode.

Remember *Test* is just a list of *inputs*.

1) If the target function consumes 1 non-list parameter, *Test* is just a list of elements. See (I)

2) If the target function consumes 1 list, *Test* is a list of list of inputs. See (II)

3) If the target function consumes more than or equal to 2 parameters, *Test* is a list of list of inputs. See (III)

# 2 Helper Functions Provided

The following is a list of useful functions I provide in “testgen.ss” and none of them are build-in functions. You should require “testgen.ss” in order to use them.

## 2.1 Create File/Directory

1)
;; create-file: String(path) String -> (void)
;; consumes a path(a String, eg. “/root/usr/” “.” “~” can be absolute or relevent path) and a String for its contents
;; Effect: create a file in that path with its contents. If the file is already there, replace it with the current one.
;; Eg:  (create-file “./1.txt” “This is its contents!”)

2)
;; create-directory: String (path) -> (void)
;; consumes a path(a String, can be absolute or relevent path)
;; Effect: create a directory based on the path. If the directory is already there, do nothing.
;; Eg: (create-directory “./”)

3)
;; copy: String String(path) -> void
;; cosumes a String (a file name with path) and a String (path)

;; Effect: copy a file into the destination path. If the file is already there, replace it with the new one.
;; Eg: (copy “1.txt” “./test/”)  (copy “./test/1.txt” “./”)

## 2.2 Convert Value to String

1)
;; l2str (list to string): (listof Any) -> String
;; convert a (listof Any) to a string using “list” notation
;; Eg: (l2str  '(1 (2) ((3)))) => “(list 1 (list 2) (list (list 3)))”

2)
;; cons2str (cons to string):
;; convert a (listof Any) to a string using “cons” notation
;; Eg: (cons2str '(1 2)) => “(cons 1 (cons 2 empty))”

3) v2str (any value to String): Any -> String
;; consumes any build-in data type (eg. Sumbol String List Posn Char ....) and produce the result in String form.
;; eg (v2str (make-posn 1 2)) => “(make-posn 1 2)”
;; (v2str #\a) => “#\a”

# 3 Autotest Generator

;; Main functions provided
;; init --- initialize
;; lang --- check and set language level
;; testgen --- Autotest Generator
;; set-conversion --- set conversion rule for unbuilt-in data type (eg convert bt-node to string)
;; reset-conversion --- reset conversion rule
;; set-require --- set require module(s) (must put module in the same directory of testgen.ss)
;; reset-require --- reset require

## 3.1 init

init is a function which is used to create/initialize the test environment.

Usage:
(init [option1] [option2])

option1 (language level):
'B --- Begining Student (default setting)
'BL --- Begining Student w/ List Abbr.
'I --- Intermediate Student
'IL --- Intermediate Student w/ lambda

option2 (forbidden functions):
(listof Symbol)  eg. '(reverse length list-ref)

**Examples:**

1) If we want to use Begining Student, then use **(init)**. Since 'B is the default setting. Or, we can use (init 'B);
2) If we want to use Intermediate Student, then use **(init 'I)**.
3) If we want to use Begining Student w/ List Abbr and we want to forbid list-ref, reverse, then use **(init 'BL '(list-ref reverse))**

**How init is implemented:**

init creates the following file system,

```
./
    answers/
    in/
        options.ss  (&)
    provided/
        banner.ss  (*)
    computeMarks
    runTests
    config.ss
```

(&) is a test setting file. By modifies (&), we can change language, requirement and time-out setting depending on option1 and option2.
(*) is optional. init may create such file depending on option2.

### 3.2 lang

**lang** is used to change language setting. You can also re-initialize everything using **init**.

Usage:
(lang)  --- it shows the current language, and pops up a language setting box asking you to set the language.
Eg.
>(lang)
Your current lang-level is scheme/beginner. Which lang-level would you want to change to (B,BL,I,IL):
 ( Type in  [ BL ] Enter!)
>

**How lang is implemented:**
lang only changes language level in ./in/options.ss.

### 3.3 testgen

testgen is used to generate test cases.

**Usage:**
(testgen question submit func testcase [option1 'non-list] [option2 0])
question (string)  ---question number  eg. “1a” “6BONUS”
 submit  (string)  ---submit file name eg. “b-tree.rkt”
 func  (function pointer)  ---function to test eg. reverse length

testcase (list)  ---A list of test cases (inputs)

option1(test generating mode):

'non-list  --- default setting, all secenrio NOT 'list

'list  --- func takes 1 param which is a list

'custom  --- custom setting

option2:

0  --- checking exact output

positive-num(eg. 0.01)  ---tolerance, (checking inexact output)

String  ---(eg. “(result (f ~a)) \n(expected ~a)”) pair with option1=='custom

**Examples:**

1) (testgen “q1” “a1q1.rkt” q1func testq1 )

question: “q1”

student submit: “a1q1.rkt”

function to test: q1func

test: testq1 (which is a list of inputs)

test generating mode: 'non-list (default)

2) (testgen “list-question” “a5q2.rkt” my-list-func testq2 'list 0.01)

question: “list-question”

student submit: “a5q2.rkt”

function to test: my-list-func

test: testq2 (a list of inputs)

test generating mode: 'list  (my-list-func consumes ONE parm,which is a list)

option2: 0.01 the result has 0.01 as tolerance.

3) (define testStr “(result (local [(define result-val (f ~a))

(define expect-val ~a)]
(equal? result-val expect-val)))
(expected true)")
(testgen "5BONUS" "f.rkt" f test6 'custom testStr)

question: "5BONUS"
student submit: "f.rkt"
function to test: f
test: test6
test generating mode: 'custom
option2: testStr
Autotest Generator will use testStr as a template. It will fill in all parameter from test to the first ~a automatically. And replace the second ~a with the expected value.

See more explanations and how it is implemented in Chapter 4 Test Generating Mode.

### 3.4 set-require/reset-require

Sometimes, test cases require some other racket files or teach packs. You need to use set-require **before** testgen, and use reset-require **after** testgen.

**Usage:**

```
;; set-require: (Union String (listof String)) -> void
;; it consumes either one single file name ,or a list of file names
;; Effect: if you put the reqired files in the same directory, those files will be
```

copied to ./provided/ automatically. And from now on, if you call testgen, it will modify ./in/QUESTION/option.ss so that each test case will require that file.

;; Example: (set-require “question1require.rkt”)

(set-require (list “require1.rkt” “require2.rkt”))

;; reset-require: void -> void

;; call (reset-require) will reset all requirement setting,ie. undo set-require. eg. testgen will not modify option.ss

Examples:

1) (set-require “assn1.rkt”)

(testgen “q1” ......)

(testgen “q2” ......)

(testgen “q3” ......)

(reset-require)

q1 to q3 all ask student to require “assn1.rkt”.

2) (set-require (list “q1a.rkt” “q1b.rkt”))

(testgen “question1” ......)

(reset-require)

(set-require “q2.rkt”)

(testgen “question2” ......)

(reset-require)

question1 requires “q1a.rkt” and “q1b.rkt”.

question2 requires “q2.rkt”.

## 3.5 set-conversion/reset-conversion

Autotest Generator **cannot** deal with unknown data types. If you ask testgen to consume a tree, graph or whatever not build-in, it may not produce results you are expecting. A good way to solve this question is to provide a list of default conversion rule, and let users to set up their own.

Autotest Generator use v2str (see Chapter 2 Helper Functions Provided) to convert any data type to string, and use that string to create test files (test.ss).

Here are a list of default conversion rules v2str uses:

1) Char (v2str #\a) => “#\\a”
2) empty (v2str empty) => “empty”
3) String (v2str “abc”) => “\”abc\”“
4) Posn (v2str (make-posn 1 2)) => “(make-posn 1 2)”
5) Symbol (v2str 'abc) => “\'abc”
6) Number (v2str 123) => “123”
7) Boolean (v2str #\t) => “true”
8) Function (v2str string-append) => “string-append”
9) List (v2str '(1 2 3))
   => “(cons 1 (cons 2 (cons 3 empty)))” (Beginning Student)
   => “(list 1 2 3)” (other language level)

You can also apply v2str to nested-list.
(v2str '((1) ((2)) (((3))))) => “(list (list 1) (list (list 2)) (list (list (list 3))))”

If you want to convert unkown data types to String or changing the default setting for v2str, you need to use set-conversion **before** testgen, and may/may

not use reset-conversion to **reset** conversion rules to default.

**Usage:**

;; set-conversion: (X->Boolean) (X->String) -> void

;; consumes a boolean predicate, and a string conversion function

;; Effect: from now on, each time testgen meets with date type X, it applies the string conversion function to it automatically. If X is in the list of default conversion rules, the new rule will override the old rule.

;; Eg. (set-conversion empty? (lambda (x) “(list)”)) . Each time testgen meets empty, it will use “(list)” as its output.

;; reset-conversion: void -> void

;; By calling (reset-conversion) , conversion rules will be set to default.

**Examples:**

1) (define-struct btnode (key left right))

```
(define (bt2str bt)
    (format “(make-bt2str ~a \n ~a \n~a)”
                (btnode-key bt) (btnode-left bt) (btnode-right bt)))
(set-conversion btnode? bt2str)
(testgen.....)
.....
```

In this case you do not need to reset conversion. Since it is all depended on your string conversion function, you need to be careful.

2)

;; a list binary tree LBT is either,

;; empty, or

```
;; (list Num LBT LBT)
(define (lbt2str lbt)
  (cond [(empty? lbt) “empty”]
        [else (format “(list ~a \n ~a \n ~a)” (first lbt) (lbt2str (second lbt)) (lbt2str (third lbt)))]))
(set-conversion cons? lbt2str)
(testgen......)
(reset-conversion)
```

In this case we do need to reset conversion. Since in other questions you may also meet with lists, that's the conly case you need to reset conversion rules.

# 4 Test Generating Mode

In 3.3 testgen, we have talk about how to use Autotest Generator. In this section we will talk about how to deal with different options, how testgen is implemented, and what does the output look like.

We have three options: 'list, 'non-list, and 'custom:
I) 'list is used when the function to test takes exactly **one** parameter, and that parameter is a **list;**
II) 'non-list takes 1) one parameter which is not a list 2) more than or equal to 2 parameters.
III) 'custom requires a template(a string)

## 4.1 Mode: 'list

Quick Guide:
(testgen “q1” “a1q1.rkt” my-list-func *Test 'list*)
Where *Test* is a list of list:
eg. (define Test (list empty
(list 1 2 3)
(build-list 2 (lambda (x) (random 10)))))
Note: if my-list-func produces inexact number, use (testgen “q1” “a1q1” my-list-func *Test 'list 0.001*) where 0.001 is the tolerance.

Example:

(init)

(testgen “q1” “a1q1.rkt” length *Test 'list*)

Output:

./in

q1/

001/

test.ss

002/

test.ss

003/

test.ss

options.ss

Test files

./in/q1/001/test.ss:

(result (length empty))

(expect 0)

./in/q1/002/test.ss:

(result (length (cons 1 (cons 2 (cons 3 empty)))))

(expected 3)

./in/q1/003/test.ss:

(result (length (cons 4 (cons 7 empty))))

(expected 2)

./in/q1/options.ss

(loadcode “a1q1.rkt”)

### 4.2 Mode: 'non-list

**Quick Guide:**

(testgen “q2” “a1q2.rkt” my-func *Test 'non-list*)

Where Test is

1) a list inputs if my-func consumes exactly 1 parameter.

eg. (define Test1 (list -1 0))

2) a list of list of inputs if my-func consumes more than or equal to 2 parameter

eg. (define Test2 (list (list 0 0)
(list 1 -1)))

**Example:**

(init 'BL)

(testgen “q1” “a1q1.rkt” add1 *Test1 'non-list* )

(testgen “q2” “a1q2.rkt” + *Test2 'non-list*)

Output:

```
./in
    q1/
        001/
            test.ss
        002/
            test.ss
        options.ss
    q2/
        001/
```

test.ss

002/

test.ss

options.ss

**Test files**

./in/q1/001/test.ss:

(result (add1 -1))

(expected 0)

./in/q1/002/test.ss:

(resule (add1 0))

(expected 1)

./in/q1/options.ss:

(loadcode “a1q1.rkt”)

./in/q2/001/test.ss:

(result (+ 0 0))

(expected 0)

./in/q2/002/test.ss:

(result (+ 1 -1))

(expected 0)

./in/q2/options.ss:

(loadcode “a1q2.rkt”)

### 4.3 Mode: 'custom

**Quick Guide:**

(define *my-*template “
(result (local [(define result (my-func *~a*))
(define expect ~a)]
(= result expect)))
(expected true)”)
(testgen “q1” “a1q1.rkt” my-func *Test 'custom my-template*)

Note: Template is a string which is used to generate test cases. It contains several “~a”'s which will be replaced by all parameters the target functions use and the result value. So use one “~a” per perameter, and one “~a” for the result value.

eg.

1) “(result (+ ~a ~a))
(expected ~a)”

+: consuems 2 parameters

2) “(result (add1 ~a))
(expected ~a)”

add1: consumes 1 parameter

3) “(result (build-list ~a ~a))
(expected ~a)”

build-list: consumes 2 parameters

4) “(result (< (abs (- (/ ~a ~a) ~a)) 0.001))
(expected true)”

/: consuems 2 parameters

**Examples:**

```
(define my-template “
(result (local [(define result (add1 ~a))
                (define expect ~a)]
          (= result expect)))
(expected true)”)
(define Test (list 0 1))
(testgen “q3” “a1q3.rkt” add1 Test 'custom my-template)
```

**Output:**

```
./in
      q3/
            001/
                  test.ss
            002/
                  test.ss
            options.ss
```

Test files

./in/q3/001/test.ss:

```
(result (local [(define result (add1 0))
                (define expect 1)]
          (= result expect)))
(expected true)
```

./in/q3/002/test.ss:

```
(result (local [(define result (add1 1))
                (define expect 2)]
```

```
        (= result expect)))
(expected true)

./in/q3/options.ss:
(loadcode “a1q3.rkt”)
```

# 5 Application—Assignment :)

**Question List:**

Q1) Normal List

Q2) List w/ inexact output

Q3) Unknown Data Type(set-conversion)

Q4) Custom Output

Q5) Forbidden Functions

Q6) Function Consumes More Than 1 Parameter

Q7) Require Teachpack/Files

Language: Intermediate Student w/ Lambda

```
#lang racket
(require “testgen.ss”)
(init 'IL)
```

**Q1: (Normal List)**

write a function called sum1, which consumes a list of Nat, and produces the sum of all the elements in that list. (submit “sum1.rkt”)

Eg. (sum1 (list 1 2 3)) => 6  (testgen mode: 'list)

Answer:

```
(define (sum1 lon)
    (foldr + 0 lon))
(define test1 (list empty
```

```
                    (list 1)
                    (list 1 2 3)))
(testgen “1” “sum1.rkt” sum1 test1 'list)
```

**Q2: (List w/ Inexact Output)**

write a function called sum2, which consumes a list of Floating Num, and produces the sum of all the elements in that list. Hint: use check-within to test. (submit “sum2.rkt”)

Eg. (sum2 (list -1.1 1.1 2.45)) => 2.45 (testgen mode: 'list with inexact output)

Answer:

```
(define (sum2 lof)
    (foldr + 0 lof))
(define test2 (list empty
                    (list 1.1)
                    (list 1.1 1.2 1.3 -1.3567)))
(testgen “2” “sum2.rkt” sum2 test2 'list 0.001)
```

**Q3: (Unknown Data Type(set-conversion))**

write a function called tree-copy, which cosumes a BT, and produces a copy of that BT ;;A BT is either empty, or (make-btnode num BT BT) (submit “tree-copy.rkt”)

Eg. (tree-copy (make-btnode 1 empty empty)) => (make-btnode 1 empty empty)

(testgen mode: 'non-list )

Answer:

```
;; A BT is one of:
```

```
;; empty, or
;; (make-btnode key left right) ,where left and right are also BTs
(define-struct btnode (key left right))

;; space: num -> " "[length= 3*(num + 1)]
;; (space 0) => "   "
;; (space 1) => "      "
;; (space 2) => "         "
(define (space num)
    (build-string (* 3 (add1 num)) (lambda (x) #\space)))

;; bt2str: BT -> String
;; (display (bt2str (make-btnode 1 (make-btnode 2 '() '()) '() '()))) =>
  (make-btnode 1
                (make-btnode 2
                             empty
                             empty)
                empty
                empty)

(define (bt2str bt)
    (local [(define (bt2str-helper t deep)
              (cond [(empty? t) (string-append (space deep) "empty")]
                    [else (format "~a (make-btnode ~a \n ~a \n ~a)"  (space deep)
(btnode-key t) (bt2str-helper (btnode-left t) (add1 deep)) (bt2str-helper (btnode-
right t) (add1 deep)))]))]
              (bt2str-helper bt 0)
))
```

```
;; Answer:
(define (tree-copy t)
    (cond [(empty? t) empty]
          [else (make-btnode (btnode-key t) (tree-copy (btnode-left t))  (tree-copy
(btnode-right t)))]))

;; Test
(define test3 (list empty
                    (make-btnode 1 empty empty)
                    (make-btnode 1 (make-btnode 2 empty empty) empty)))

(set-conversion btnode? bt2str)
(testgen “3” “tree-copy.rkt” tree-copy test3)
(reset-conversion) ;; this line is optional, if you still need to use btnode, remove
this line
```

Q4: (Custom Output)

Write a function called subsets1, which consumes a list of Nat and produces a list of all of its subsets. (submit “subset1.rkt”)

Eg. (subsets1 ’(1 2)) => (list ’(1 2) ’(1) ’(2) ’()). (testgen mode: 'custom) (from Fall2013CS135 A10 BONUS)

Answer:

```
(define (subsets1 lon)
    (foldr (lambda (num accu) (append accu (map (lambda (z) (cons num z))
accu))) (list empty) lon))
```

```
;;Test
(define test4 (list empty
                    (list 1)
                    (list 1 2 3)
                    (build-list 10 add1)))


(define template4 "
(result (local [(define result-ans (subsets ~a))
                (define expect-ans ~a)
                (define (lists-equiv? l1 l2)
                  (and (= (length l1) (length l2))
                  (andmap (lambda (x1) (ormap (lambda (x2) (equal? x1 x2)) l2))
l1)
                  (andmap (lambda (x2) (ormap (lambda (x1) (equal? x1 x2)) l1))
l2)))]
        (lists-equiv? result-ans expect-ans)
))
(expected true)
")

(testgen "4" "subsets1.rkt" subsets1 test4 'custom template4)
```

Q5: (Forbidden Functions)

write a function called my-reverse, which consumes a list of any value, and produce that list in reverse order. You cannot use reverse. (submit "my-reverse.rkt")

Eg. (my-reverse (list 1 2 3)) => (list 3 2 1)(set up forbidden functions)

Answer:

```
(define (my-reverse lst)
(reverse lst))

(define test5 (list empty
                    (list 1)
                    (list 1 2 3 4)))

;; re-initialize
(init 'IL '(reverse))
(testgen “5” “my-reverse.rkt” my-reverse test5 'list)
```

**Q6: (Function Consumes More Than 1 Parameter)**

Consider the following predicate function that consumes three Booleans and produces a Boolean:

```
(define (cond-mystery? a b c)
        (cond
            [(not a) c]
            [else b]))
```

Write the scheme function bool-mystery? so that it is equivalent to cond-mystery? Except that it uses only a Boolean expression (i.e.: it does not have a cond expression). (submit “bool-mystery.rkt”).

Answer:

```
;; Note: in order to test bool-mystery, you need to plug in all possible boolean values into a, b, and c.
```

```
(define (bool-mystery a b c)
    (or (and (not a) c) (and a b)))

(define test6 (list (list #\t #\t #\t)
                    (list #\t #\t #\f)
                        (list #\t #\f #\t)
                        (list #\t #\f #\f)
                        (list #\f #\t #\t)
                        (list #\f #\t #\f)
                        (list #\f #\f #\t)
                        (list #\f #\f #\f)))
;; test6 = '((t t t) (t t f) .....(f f f))  8 possible ouputs
;; using for-loop will save you a lot of time if you have more then 3 boolean
outputs
;;eg.
;; (define test6 '())
;; (for ([a (list #\t #\f)])
;;     (for ([b (list #\t #\f)])
;;         (for ([c (list #\t #\f)])
;;             (set! test6 (cons (list a b c) test6)))))

(testgen “6” “bool-mystery” bool-mystery test6 )
;; testgen: 'non-list is the default setting
;; or (testgen “6” “bool-mystery” bool-mystery test6 'non-list)
```

Q7: (Require Teachpack/Files)

Write a function called my-lcm (least common multiple) which consumes 2

parameters and produces the least common multiple. Note: (my-lcm m n) = (/ (* m n) (my-gcd m n)). my-gcd will provide to you in “my-gcd.rkt”.

Answer:

```
(require “my-gcd.rkt”)
(define (my-lcm m n)
(/ (* m n) (my-gcd m n)))

;; Test
(define test7 (list (list 5 10)
                    (list 3 7)
                    (list 32767 1024)))
(set-require “my-gcd.rkt”)
(testgen “7” “my-lcm.rkt” my-lcm test7)
(reset-require)
```