

Racket - P1

racket : the core compiler, interpreter, and run-time system

DrRacket : the programming environment (编程环境)

reco : a command-line tool for executing Racket commands that install packages, build libraries, and more

#lang 在 DrRacket 中, 通过 #lang 选择 proper language, 也可以通过 Language | Choose Language

DrRacket's bottom text area 和 racket command-line program 表现如 calculator

REPL 即输入 Racket 表达式, 即时输出结果, 比类 calculator 被称为 real-eval-print loop

可以看作一个不断等待输入表达式并计算表达式结果及输出的循环 (类似于 C++ 中实现的计算器)

expression 在 Racket 中, 几乎所有类型的 expression 都需要用 () 括起来, 包括定义的部分

函数调用 在 Racket 中函数以如下形式调用: (function name argument expression), 其中可有多个 argument

definition 通常位于 DrRacket's top text area, 也称 (the ~~area~~ definition area)

在 definition area 中的表达式, 在点击 DrRacket 的 Run 时进行 evaluation 以及输出

Executable : 在 DrRacket 中, 可以通过 Racket | Create Executable 将 Racket source 文件变为 .exe 文件

Values Racket 中的 value 包括 Number, Boolean, String, 和 Byte String

Number : 包括 整数, 小数, 分数 (fraction), 科学计数法 (如 6.02e+23), 虚数 (1+2i)

注意 整数属于无限精度 Integer, 即可以输入大于 $2^{31}-1$ (2147483647) 的整数

Boolean : #t 表示 True, #f 表示 False, 另外所有非 #f 的值都被视为 #t

String : 定义在 " " (double quote) 之内, 反斜杠 \ (backslash) 的用法与 C++ 中类似

任何 Unicode 字符都可以定义在 String 中, (除了 unescaped double quote)

Print (输出) 当一个常量在 REPL 中计算时, 通常来说 input syntax, 但某些情况下输出

某些情况下, 会输出一个通常化的版本 (normalized version of input syntax)

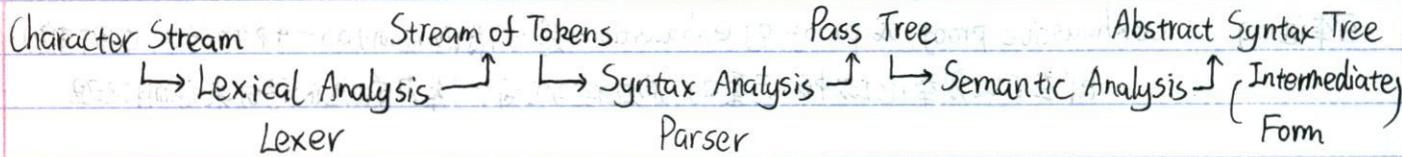
input expression 用绿色显示, printed result 用蓝色表示

comment 在 definition area 中, 用 ; 表示备注 (comment), 位于 ; 之后到该行结束的部分均被视为备注

Racket - P2

program
module

Racket 有 program module (及其他 definition) 类似 EBNF (Backus-Naur Form)
在 definition area 中写为 #lang <lang name> <top form>*
注意在 syntax specification 中 #lang 属于 literal text, 还包括 () [] 等符号
<...> 表示 nonterminal, 在其与 literal 之间必须用 whitespace 隔开, 除了 () 和 []
* 在 grammar 中表示元素出现 0 次或若干次, 即 $a^* \rightarrow \lambda, "a", "aa", \dots$, 出现 k 次, $k \in \mathbb{N}$
+ 在 grammar 中表示元素出现 1 次或若干次, 即 $a^+ \rightarrow "a", "aa", \dots$, 出现 k 次, $k \in \mathbb{Z}^+$
{} 在 grammar 中表示扩号中的部分 repeat k 次, $k \in \mathbb{N}$, 即 $\{abc\}^k \rightarrow "\lambda, "abc", "abca", \dots$



Formal
Language

- An alphabet of symbols (set of characters) Σ $\Sigma = \{a, b\}$
- A language over a set of symbols $L_1 = \{aa, bb\}, L_2 = \{\lambda, aa, aaa, \dots\}$
- L is a set of sequences built from these symbols

Regular
Language

over an alphabet Σ :

- The empty language \emptyset is regular
- The finite language $\{a\}$ where $a \in \Sigma$ are all regular
- The language containing the empty word λ is also regular, i.e. $\{\lambda\}$
- Given two regular languages L, M then
 - $L \cup M$ is regular (union)
 - $L \circ M$ is regular (concatenation)
 - L^* is regular (kleene Star)
 - other languages aren't regular

Regular
Gramma

Set of Rules for generating words in a language

N : Set of non-terminal symbols

Four components Σ : set of alphabetic characters for Language L

(N, Σ, P, S) P : Set of production rules

S : Start Symbol

定义语法 $G = (V, N, S_0, P)$

N 为语义 G 的非终结符集合

V 为语义 G 的符号集. $V = N \cup T$

P 为语义 G 的产生式集合 $\{f: w_1 \rightarrow w_2\}$

S_0 为语义 G 的起始符 正则: $A \xrightarrow{a} ab$

Racket - P3

Finite Automata Five components $(Q, \Sigma, \delta, q_0, F)$
State Machine
有限自动机
 Q is set of states
 Σ is the alphabet for the language
 δ is a transition function $\delta: (Q, \Sigma) \rightarrow Q$
 q_0 is the starting state
 F is the set of final or accepting state

定义有限自动机 $M = (S, \Sigma, f, s_0, F)$
 S 为有限状态自动机 M 的状态集。
 Σ 为 M 的输入字符集。
 f 为转移函数, 有 $f(S, \Sigma) \rightarrow S$
 s_0 为 M 的起始状态
 F 为 M 的终结状态集。

Regular Expression
正则表达式
 $\lambda, \text{Var } [a], \emptyset$ is Regular Expression
 R, S are Regular Expressions
 $R \circ S$ concatenation
 $R | S$ alternation or "choice"
 R^* Kleene Star
 (R)

空集 \emptyset , 空串 λ , $\text{Var } [a]$ 是正则表达式 基础步骤
若 R, S 是正则表达式集合
则 RS (连接)
 $R|S$ (并集)
 R^* (R 的克莱因闭包)
 (R) 括号 (R) 是正则表达式集合 递归步骤

Definition Racket 中存在两种形式的定义, 分别对应 变量或值 (value) 和 定义函数 (definition), 都在 area 中
 $(\text{define}, \langle \text{id} \rangle \langle \text{expr} \rangle)$ 用于 bind $\langle \text{id} \rangle$ to the result of $\langle \text{expr} \rangle$, 如 $(\text{define} \text{ pie} 3)$
 $(\text{define} (\langle \text{id} \rangle \langle \text{id} \rangle^*) \langle \text{expr} \rangle^+)$ 用于 bind 1st $\langle \text{id} \rangle$ to a function (Racket 中也称 Procedure)
function name argument function body (last is return) take arguments as named by the rest $\langle \text{id} \rangle^*$
如 $(\text{define} (\text{piece} \text{ str}) (\text{substring} \text{ str} \text{ 0} \text{ pie}))$ function body as $\langle \text{expr} \rangle^+$ (至少有 1 个 $\langle \text{expr} \rangle$)
return result as the last $\langle \text{expr} \rangle$ of $\langle \text{expr} \rangle^+$

注意: 理论上 function definition 与 non-function definition 是一样的。

function name 也不一定作为函数调用使用, 如 > piece → #<procedure: piece>
function 本身也是一种 value

Side-effect 在 function definition 中, 允许存在多个 $\langle \text{expr} \rangle$, 但只有最后一个 $\langle \text{expr} \rangle$ 的值作为函数值返回
副作用 而其他 $\langle \text{expr} \rangle$ 仅仅计算用于 side-effect, 如 function body 出现的 $(\text{printf} "...")$ 表达式
注意: 如果上例中函数错误定义为
考虑其过程为 $\langle \text{expr} \rangle_1 (\text{substring}) \rightarrow \#<\text{procedure: substring}>$
 $(\text{define} (\text{piece} \text{ str}) \langle \text{expr} \rangle_2 \text{ str} \text{ "hello"}) \rightarrow \langle \text{expr} \rangle_2 \text{ str} \text{ "hello"}$

则 > $(\text{piece} \text{ "hello"}) \rightarrow 3$, (预期为 "hel")

$\langle \text{expr} \rangle_3 \text{ 0} \rightarrow 0$

$\langle \text{expr} \rangle_4 \text{ pie} \rightarrow 3$ (return result)

可见未在一个括号中的部分, 将被视为不同表达式, 而分别 evaluate value

而由于函数名本身可以被当成 value 使用, 所以编译器和运行时均不会报错

但是有括号时, () 在 grammar 中表示 group, 则 function name 会被当成 function call,

此时如果此时参数数量不对或类型不对, 则会在运行时报错 arity mismatch / contract violation

Racket - P4

Identifier 在 Racket 中，标识符的语法 (Syntax) 是非常宽松自由的。如常量 (number constant) (sequence) 标识符 除了 特殊字符 () , [] , { } , " " , ' ' , ` ' , , , ; , # , | \ 以及形成数字常量的字符串序列
几乎所有的 非空白符 (non-Whitespace) 非空字符 的字符都可以 组成 `<id>`
注意：如 $a+b$ 通常被视为算式 (arithmetic expression)，在 Racket 中会被视为 Identifier 即 $> 1 + 2 \rightarrow 1 \ #<\text{procedure} : + > 2$ 即以空格隔开被视为 3 个 value

Function Call 也称 (Procedure Application)，语法结构形如 `(<id> <expr>*)`，注意必须要有括号 ()
函数调用 语法中 () 表示一个 group，其中 `<id>` 将被用作 function name，非函数则报错 not a procedure
`<expr>*` 用作 arguments supplied to the function name `<id>`。
如 $(+ 1 2) \rightarrow 3$ ，注意 $(1 + 2)$ 报错由于 1 无法作为 procedure。
而 $(1 + 2)$ 报错 undefined 由于 1+2 被视为 Identifier
注意：可以用如 `(define (+ a b) (+ a b))` 将 $(+ a b)$ 定义为与加法 等价

Table of Extended Backus-Naur Form (EBNF) 中的 特殊符号表 (ISO/IEC 14977) (*...*) comment										
Symbol	=	,	; /.		-	(...)	[...]	*{...}	;"..."	? ... ?
definition				termination	alternation	exception	grouping	optional	repetition	terminal
				非终结符 / 语法	definition					特殊符号 / K star

注意 Racket 语法与 EBNF 并非完全一致，但是相似部分较多。

if conditional 的语法结构为 `(if <expr> <expr> <expr>)`

其中第一个 `<expr>` 总是 evaluated，只结果为 non-#f，evaluate 第 2 个，否则 evaluate 第 3 个
特别注意 Racket 对 non-#f 的认定非常宽松，如 $(if + 1 2) \rightarrow 1$ ，procedure 也是 non-#f

and 形如 `(and <expr>*)`，and 遵循 short-circuit；stop and return #f when an `<expr>` produce #f

or 形如 `(or <expr>*)`，相似的 short-circuit；stop and return value when an `<expr> \rightarrow` non-#f

注意：在 Racket 中，0 不被视为 #f，即 0 是一个 non-#f value

当 and 和 or stop and return 时，如果当前 `<expr>` 是 non-#f value，则返回 value 而非 #t

cond 形如 `(cond { <expr> <expr>* }*)`，有 a sequence of clause between square bracket
对于每一个形如 `[<expr> <expr>*]` 的 clause
`(cond [(equal? "hello" s) "hi!"]` 如果第一个 `<expr>` 是 non-#f value 则返回 最后一个 clause
`[(equal? "bye" s) "bye!"]` 一个 `<expr>` 的 value (可以返回第一个)
`[else "?"])` 如果第一个 `<expr>` 是 #f，则继续下一个 clause
else 视为 synonym for #t text expression

Racket - P5

- (), [] 原则上 Racket 中 () 和 [] 是等价的, 只要对应即可, 只是符号语法的 code 更 readable
- 在 Function call 中, 可以使用形如 (<expr> <expr>*) , 只要符合两个条件:
1. 第一个 <expr> 的 value 可以被视为 <id> 且有相应的 <procedure>
 2. 后续的 <expr>* 的 values 符合第一个 <procedure> & arguments 的数量和类型
- 如 ((if (> 1 2) + -) 1 2) → -1, 其中 if 表达式返回了一个 Identifier, 并用作 function call

collatz (define (collatz-e n) (define (collatz (n) 注意在右侧的 if 表达式
(/ n 2)))
(define (collatz-o n) (if (equal? 0 (modulo n 2)) 根据 n 是奇数或偶数
(+ 1 (* 3 n))) collatz-e)
collatz-o (n)))
function call

lambda 在 Racket 中, lambda 的 syntax 为 (lambda (<ids>*) <expr>+), 其 value 为一个 procedure
<ids>* 为函数的参数, 注意可以为 0 个, 即不需要传入参数, 但是括号依旧要写, 如 (lambda () 1)
<expr>+ 为函数体, 返回值为最后一个 <expr> 的 value
所有的 define(<id> <expr>) 都可以视为 non-function definition 是因为 <expr> 可以是 lambda
即 (define (collatz-e n) (define (collatz (e 是等价的, 而后者的形参
(/ n 2))) (lambda (n) (/ n 2)))) (define <id> <lambda>) 定义
另外 (define (times n) 中返回的是一个 lambda-generated function, "remember" 正确的 n
(lambda (x) (* n x))) 因为 Racket 是一种 lexically scoped language

lexically scoped (词法作用域) 指标识符被固定在被定义的作用域上, 而非语法或其被调用的作用域

Internal definition Racket 中允许在定义函数参数和函数体之间插入其他 definition
即形如 (define (<id> <id>*) <definition>* <expr>+) 或 (lambda (<ids>*) <definitions>* <expr>+)
考虑其作用, 主要是定义函数中使用的常量, 避免在函数中使用魔法常量,
另外是定义函数中可能使用的子函数, 使得函数代码更加简单

let / let* Racket 中的 let 与 Haskell 相比, 区别在于 let* 并加以区分。
有形如 (let ([<id> <expr>]*)<expr>+) 和 let* ([<id> <expr>]*)<expr>+
在每个 clause 中都有将 -> <expr> 的 value (绑定到) 一个 <id>, let* 中可以使用之前 clause 中定义的 <id>
如 (let* ([a 1] [b (+ a 3)]) (+ a b)) → 5 而 let 只能分别定义, 可见 let* 与 Haskell 中的 let 语法是相似的

Racket - P6

Regular Language

languages can be described using Regular Gramma, Deterministic Finite Automata, Regular Expression
if and only if (当且仅当) 正则语法 确定有限状态自动机 正则表达式

Chomsky Hierarchy

regular) context-free) context-sensitive) recursively enumerable

• 3型(正则文法) 2型(上下文无关文法) 1型(上下文有关文法) 0型文法

Lisp

Racket 是 Lisp 的一种 dialect (方言), Lisp 源于 "LIST Processor"

list

Racket 中, list 的 print form (显示格式) 为 '(<value>*), 如 '(1 2 3) → '(1 2 3)

list 同样可以直接以这种形式输入, 注意在这种形式中不计算 <expr> 的 value

list 也可以通过 list 方法生成, 其语法为 (list <expr>*), 如 '(list 1 2 3) → '(1 2 3)

注意 () (parenthesis) 在两种语法中具有不同的意义, 在 '(<value>*) 中 () 不再被视为代表 group

在 list 方法中, element 被视为 <expr>, 在组成 list 前先 evaluate value,

如 (list (+ 1 1) 2 3) → '(2 2 3), 即 (+ 1 1) 被视为 <expr>, value 为 2

在 '() 形式中, element 被视为 value, 或者递归地视为嵌套 list, 即语法为 '({<value>} | list)*

如 '((+ 1 1) 2 3) → '((+ 1 1) 2 3). 即 (+ 1 1) 被视为下一层 list, value 为 '+ 1 1'

由此可看出, Racket 的 list 也不要求 list 的元素为同一类型, 特别注意 procedure 可作为 value

可以将 Racket 的 list 看作是一个链表.



如 '((+ 1 1) 2 3) 可视为:



特别注意: 当使用 (let ([a (+ 1 1)]) '(a 2 3)) 生成 list 时, 生成 list 为 '(a 2 3)

其中存放的为在 let 中 bind expression (+ 1 1) 的 Identifier a,

即使用 list-ref 取出 a, 不会计算 <expr> 的 value, 返回值为 a

length

用于计算 list 中的元素, 调用形式为 (length list), 注意只计算 list 第一层中的元素个数,

list-ref

用于返回 list 中指定 index 的元素, 调用形式为 (list-ref list index), 如果 index 超界返回 large for list

注意只计算 list 当前层中的元素, 如果指向的元素是 list, 则返回值为 list

member

用于查找 list 中是否存在元素 value, 调用形式为 (member value list), 注意只查找当前层

即 P (member 2 '(1 2) 3 4)) → #t, 如果找到, 则返回该元素及之后的部分

类似于 dropWhile 函数

reverse

用于反转 list, 返回一个 list 但元素顺序相反, 注意也只对当前层反转, 调用形式 (reverse list)

(P. AP (reverse '(1 2) 3 4)) → '(4 3 '(1 2))

Racket - P7

append 用于连接两个 list, 例用形式为 (append list1 list2), 特别注意只添加一个元素的情形
即有 (append '(1 2) '(3)) \rightarrow '(1 2 3), 但 (append '(1 2) 3) \rightarrow '(1 2 . 3)

list-iteration functions iterate over elements of a list, Racket iteration packaged into function function will be applied to each element, lambda form in combination with iteration functions

这一类函数与其他语言中的高级函数 (map, filter) 相似, 相互之间 combine iteration results in different ways

map 与 Haskell 中 map 类似, 例用形式为 (map <procedure> list)

use pre-element results: & so (map sqrt '(1 4 9)) \rightarrow '(1 2 3). 注意元素的类型是否符合函数

andmap 用于检查 \exists list p1, ..., \exists list pn, combine results by anding ororing

ormap 例用形式如 (andmap / ormap <procedure> list), 注意 <procedure> 有 $a \rightarrow \text{Bool}$ 的类型
andmap 类似 Haskell 中的 (foldl (&&) True), (map), ormap 有 (foldl (||) False), (map)

传入多个 list map, andmap, ormap 可以传入多个列表, 即有例用形式 (andmap / ormap ($\rightarrow \text{Bool}$) {list})

要求所有 list 必须有相同长度, 即 Racket 中的 map 包含了 Haskell 中的 zip, zip3, zip4 ...

要特别注意函数 ($\rightarrow \text{Bool}$) 会依次传入每个 list 的一个元素, 注意 list 的顺序和类型

& so (map list '(1 2) '(2 4) '(4 8)) \rightarrow '((1 2 4) (2 4 8))

filter keep element when body result is true, discard when #f, 注意 non-#f value 都视为 #t
与 Haskell 中的 filter 类似, 例用形式为 (filter <procedure> list), <procedure> 有 ($a \rightarrow \text{Bool}$)

foldl use pre-element function - process element - combine with "current" value, starting "current" must be provided

例用形式为 (foldl f x, l \rightarrow l list), 特别注意与 Haskell 中 foldl 的区别在于 f 中参数的顺序

Racket 中 f = $\lambda x \rightarrow \lambda l \rightarrow l$, 而 Haskell 中顺序相反, 即 f = $\lambda l \rightarrow \lambda x \rightarrow l$

& so (foldl clambda (x l) (+ l (* x x))) on '(1 2 3) \rightarrow 14.

lambda 参数 特别注意 lambda 定义时如果只有一个参数 x, 也必须加括号如 (lambda (x) <expr>)

mcons mutable cons

creates cells which can be mutated

first Racket 中的 list 是 linked list (链表), 所以 first, rest 用于得到 non-empty list 的 tail

rest & so (first '(1 2 3)) \rightarrow 1, (rest '(1 2 3)) \rightarrow '(2 3)

(Haskell)
head
tail

Racket - P8

empty 用于生成一个空 list 作为 "construct list" 的起始，与 '()' 等价，类似于 Haskell 的 []

cons cons 用于生成列表时与 Haskell 中的 (:) 类似，即 list 可以看成是 cons 的一系列嵌套调用
'(1 2 3) 与 (cons 1 (cons 2 (cons 3 empty))) 与 (1 : (2 : (3 : []))) 是等价的

cons? cons? 用于判断是否为非空 list (推广来看即判断是否有 cons)，empty? 用于判断是否为空 list

tail-recursion 在 Racket 中，同样存在 tail-call optimization，guarantee about the way the code will run
so (define (length lst)

在调用 (length (list 1 2 3))

(define (citer lst len)
(cond [(empty? lst) len]
[else (citer (rest lst) (+ len 1))])
(citer lst 0))

= (citer (list 1 2 3) 0)

= (citer (list 2 3) 1)

= (citer (list 3) 2)

= (citer (list) 3) = 3

also (define (map f lst) (for/list ([x lst]) (f x)))

(define (citer lst result)) 注意在 Racket 中有列表生成式

(cond [(empty? lst) (reverse result)] (define (map f lst))

[else (citer (rest lst) (cons (f (first lst)) result))]) 在展开时会有与 (citer) 相同的代码

(citer lst empty) 差别仅是 Syntactic convenience

iteration 在 Racket 中，认为 iteration 是 recursion 的一种特殊形式，即形如 tail-recursion

通常来说，应该应用 tail-recursion 来避免 O(n) space consumption 情况，以及 stack overflow

但 Racket 中实际上没有 stack overflow，可以 run out of memory if a computation involves too much
but exhausting memory typically requires orders of magnitude deeper recursion (更深层次)

tail-recursion program automatically run the same as a loop (尾递归)

so (define (remove-dups l))

在调用 (remove-dups (list 1 2 2 2 3)) 时

(cond [(empty? l) empty]
[(empty? (rest l)) l]
[else (let ([i (first l)])
 (if (= i (first (rest l)))
 只有 discard
 = (cons i (remove-dups (list 2 2 3)))
 忽略 tail
 recursion
 = (cons i (remove-dups (list 2 3)))
 = (cons i (cons 2 (remove-dups (list 3))))
 = (cons 1 (cons 2 (remove-dups (list 3))))
 = (cons 1 (cons 2 (list 3))))]))])

从一个已排序的列表中

(remove-dups (rest l))

= (cons 1 (cons 2 (list 3)))

删除相同元素

(cons i (rest l))

注意虽然程序是 iteration (tail-recursion)

类似于 Haskell 中的 hub

(remove-dups ())))]))])

但由于 cons 等价于生成，所以也 O(n) space consumption

Racket - Pg

pair

在 Racket 中, cons 的结果并非总是 list, 更通常的情况是生成一个 Pair

当 cons 的第二个参数 argument 不是 empty (空列表) 也不是其自身时, 会以另一种形式输出
如 $(\text{cons} \ V_1 \ V_2) \rightarrow '(\!V_1 \ . \ V_2)$, $(\text{cons} \ 1 \ 2) \rightarrow '(\!1 \ . \ 2)$, 注意中间是 α period surrounded by white space

相比列表 list, pair 有 more traditional function 用于 non-list pair

pair? (对应于 cons?), car (对应于 first), cdr (对应于 rest , 读作 "could-er")

to make-hash 函数生成 list of pairs, car 是一个 key, cdr 是一个 arbitrary value

注意: 在打印 pair 的第二个元素是 non-list pair 时, 其形式例如 $(\text{cons} \ 0 \ (\text{cons} \ 1 \ 2)) \rightarrow '(0 \ . \ 1 \ . \ 2)$

其规则是: 1. 使用 dot notation, 除非 dot 后紧跟一个左括号 (open parenthesis)

2. 在上述情况下, 删掉 dot, 左括号 (open parenthesis) 和匹配的右括号 (close parenthesis)

如 $'(0 \ . \ 1 \ . \ 2) \rightarrow '(0 \ 1 \ . \ 2)$, $'(0 \ . \ 1 \ . \ 2 \ . \ (2)) \rightarrow '(0 \ 1 \ 2) + 2$

quote

在 Racket 打印 list 时, 有一个 quote mark “'” 在列表之前, 但是 inner list 之前不用加 quote mark

如 $(\text{list} \ (\text{list} \ 1) \ (\text{list} \ 2 \ 3) \ (\text{list} \ 4)) \rightarrow '(\!(1) \ (2 \ 3) \ (4))$

对于嵌套列表 (nested list), quote 允许直接将 list 打印成 list 打印的形式, 方便编程

即 $(\text{list} \ (\text{list} \ 1) \ (\text{list} \ 2 \ 3) \ (\text{list} \ 4)), (\text{quote} \ ((1) \ (2 \ 3) \ (4))), '(1) (2 \ 3) (4))$ 等价

特别注意 quote 可以与 dot notation 一起使用, 且不论是否使用 dot-parenthesis elimination rule

符号 (Symbol) 是一个形如 ' id ' 的 quoted identifier, 类似于加 quote marked 的符号表示, 与 expr 区分

特别注意一个 ' id ' 是 symbol 和原来的 ' id ' 只拥有相同的字母, 但不会做 ' id ' 预先定义的事

另外可用 string->symbol 和 symbol->string 将 symbol 和 string 相互转化

注意 quote 会将内嵌列表 (nested list) 都自动 quote, 且由于最外层的 '' (..) 使得内部 symbol 没有 ''

如 $(\text{quote} \ (\text{road} \ \text{map})) \rightarrow '(\!\text{road} \ \text{map})$, $(\text{ccar} \ '(\!\text{road} \ \text{map})) \rightarrow '\!\text{road}$

特别注意这种写法 ''(<ids>+) 不能用于表示值的 list, 因为其中的 <ids> 都自动成为 symbol

constant quote 不作用于常量, 如 $(\text{quote} \ "hello") \rightarrow "hello"$

简写 '' 在 Racket 中可以直接在需要 quote 的 form 前添加符号 '' (quote mark) 以表示 quote

在 DrRacket 中语法上会自动将单引号 '' 改写为 (quote ...) 的形式

如 ''road 与 '(quote road), (quote (quote road)) 是等价的, 结果均为 "road"

而 (car ''road) \rightarrow (car '(quote road)) \rightarrow 'quote

所以要特别注意 quote 和单引号的使用

Racket Syntax

Racket 的 Syntax 不是直接定义在字符流 (character stream) 上, 而是分为两层:

reader layer 读入一个字符序列并转化为列表, 符号 (symbol), 常量 (constant) 和

expander layer 将列表, 符号, 常量 描述为表达式

注意 (+ 1 . (2)) 在 Racket 中合法是因为语法上会删除 dot 和括号, 使其成为 (+ 1 2)