NKS Experimental Design and Implementation for Students

Jamie Raymond
Northeastern University
raymond@ccs.neu.edu

An NKS 2004 Presentation




Introduction

- Experimentation is at the heart of NKS.
- Conducting general NKS experiments involves programming.
- Where can NKS and programming meet at the pre-college level?

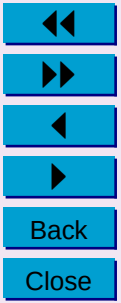



Pre-college

Two approaches to introducing NKS

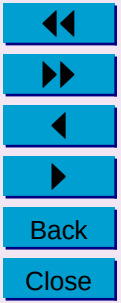
1. Computing courses: interesting problem domain
2. Domain courses (science,math):
   computer experiments that require programming

My Experience



Roadmap

- NKS Experimental Design
- Programming for NKS Experiments
- TeachScheme!
- Introductory NKS Exercise



NKS Experiments

Simple Programs

- CAs, Turing machines, substitution systems,
- numerical systems, multiway systems, and et cetera.

Experimentation

- Data generation
- Analysis (visual, statistical, etc.)




Back
Close

Scientific Method for NKS

Textbook Scientific Method

1. Observation
2. Hypothesis
3. Experiment
4. Conclusion




Scientific Method for NKS

NKS Scientific Method

1. Observation
2. Question
3. Visualization
4. Experiment: Generation and Analysis
5. Conclusion




Beginning Programmers

Cognitive Model

Issues

- Syntax
- Errors
- Blank screen




Programming

- Language
- Environment
- Methodology

One language fits all?

- C?
- Java?
- Python?
- Scheme?
- Mathematica?




TeachScheme!

Less syntax, more problem solving

Pillars of TeachScheme!

- Curriculum
- Language Levels
- Programming Environment




Curriculum

- *How to Design Programs*

  Felleisen, Findler, Flatt, Krishnamurthi

  MIT Press, 2000

- `http://www.htdp.org`




Design Recipe

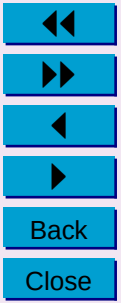
1. Problem Analysis & Data Definition
2. Contract, Purpose, Header
3. Examples
4. Function Template
5. Function Definition
6. Tests




Template matches Data Definition

```
A List-of-Number is one of:
  -- empty
  -- (cons Number List-of-Number)

;;Template
;;LON-f: List-of-Number -> ???
(define (LON-f a-lon)
  (cond
    [(empty? a-lon) ...]
    [(cons? a-lon) ...(first a-lon) ...
                   ...(LON-f (rest a-lon))]))
```



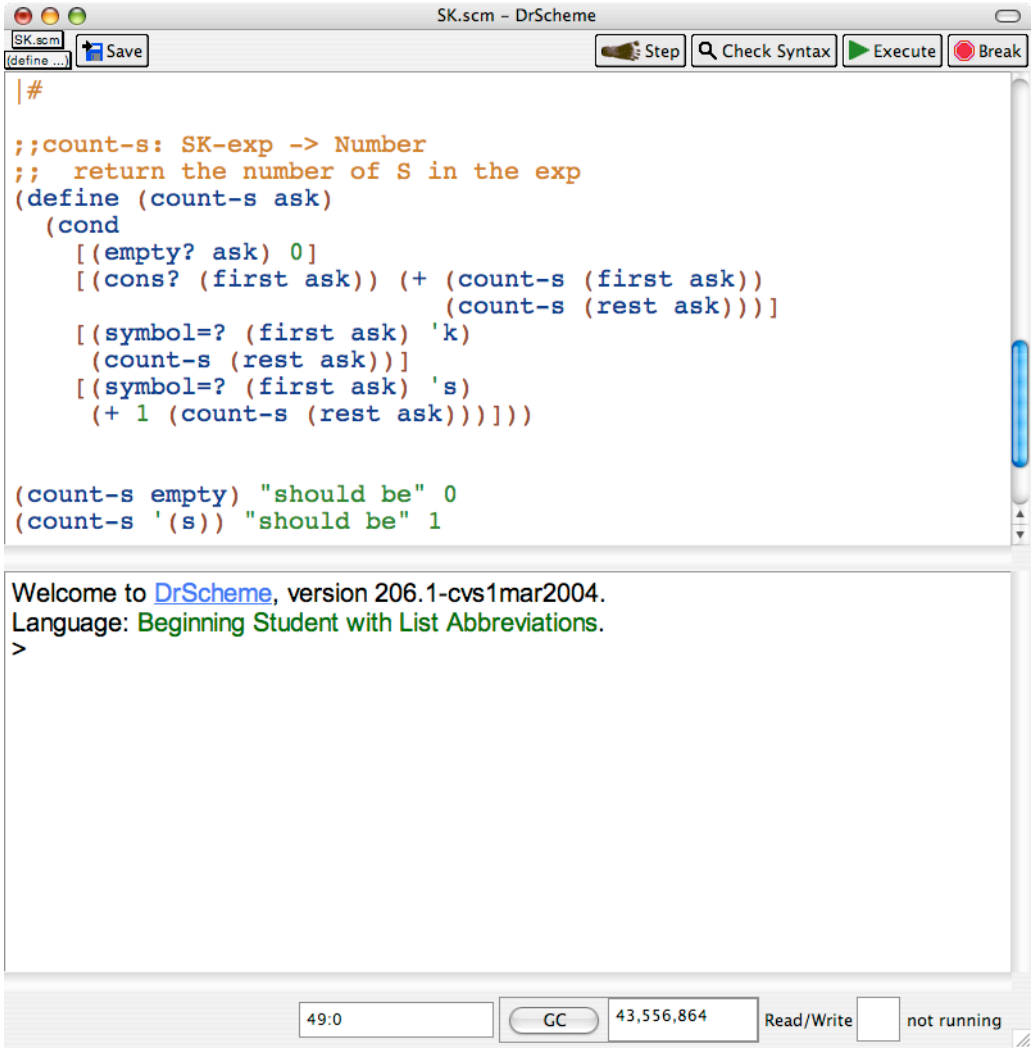
Language Levels

- Beginner through Advanced
- Better error reporting
- More pedagogic control
- Teachpacks




Back
Close

Programming Environment: DrScheme


SK.scm - DrScheme
SK.scm
(define ...)
Save
Step
Check Syntax
Execute
Break
|#
;;count-s: SK-exp -> Number
;; return the number of S in the exp
(define (count-s ask)
(cond
[(empty? ask) 0]
[(cons? (first ask)) (+ (count-s (first ask))
(count-s (rest ask)))]
[(symbol=? (first ask) 'k)
(count-s (rest ask))]
[(symbol=? (first ask) 's)
(+ 1 (count-s (rest ask)))]))
(count-s empty) "should be" 0
(count-s '(s)) "should be" 1
Welcome to DrScheme, version 206.1-cvs1mar2004.
Language: Beginning Student with List Abbreviations.
>
49:0
GC
43,556,864
Read/Write
not running

DrScheme

- Interactive
- Algebraic Stepper
- Syntax Checker
- Errors Matter!




Back
Close

Introductory NKS Exercise

Analyze an arbitrarily complicated S-K combinator expression and produce the the number of Ss in the expression.

Extensions

- Statistics (e.g. distribution of S and K)
- Generation (create raw data)
- Compilation (combinators emulating other systems)




S-K Combinators

- Symbolic system (see NKS, pg 711)
- Two rules:

```
s[x_][y_][z_] -> x[z][y[z]]
k[x_][y_] -> x
```

- `s[s[k[s]][s[k[s[s[k][k]]]]][s[k[k]][s[s[s[s[s[k][k]]]]]]]`




Data Definition

```
An SK-Exp is one of:
  empty
  (cons 's SK-Exp)
  (cons 'k SK-Exp)
  (cons SK-Exp SK-Exp)
```




Contract/Purpose/Header

```
;;count-s: SK-exp -> Number
;;  return the number of S in the exp
(define (count-s ask) ...)
```




Back
Close

Examples

```
(count-s empty) "should be" 0
(count-s '(s)) "should be" 1
(count-s '(s k s)) "should be" 2
(count-s '((s s k (s (s s k))) k)) "should be" 5
```




Back
Close

Function Template

Reminder: Data Definition

```
An SK-Exp is one of:
  empty
  (cons 's SK-Exp)
  (cons 'k SK-Exp)
  (cons SK-Exp SK-Exp)
```

Template

```
(define (SK-fun ask)
  (cond
    [(empty? ask) ...]
    [(cons? (first ask)) ...(SK-fun (first ask))
                         ...(SK-fun (rest ask))...]
    [(symbol=? (first ask) 's)
      ...(first ask) ...(SK-fun (rest ask))...]
    [(symbol=? (first ask) 'k)
      ...(first ask) ...(SK-fun (rest ask))...]))
```

Function Body

```
(define (count-s ask)
  (cond
    [(empty? ask) 0]
    [(cons? (first ask))
       (+ (count-s (first ask)) (count-s (rest ask)))]
    [(symbol=? (first ask) 'k) (count-s (rest ask))]
    [(symbol=? (first ask) 's)
     (+ 1 (count-s (rest ask)))]))
```

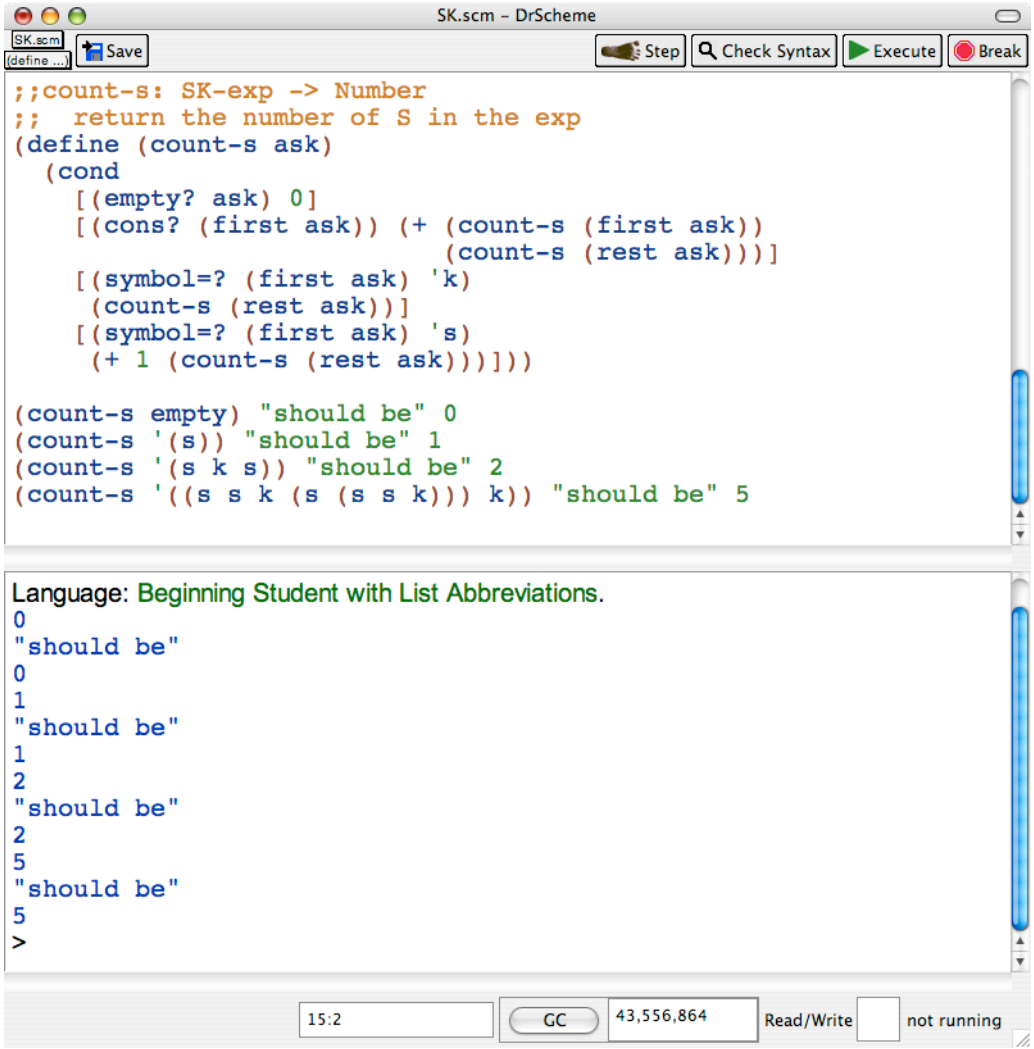



Back
Close

Testing

Back

Close


SK.scm - DrScheme
SK.scm
(define ...)
Save
Step
Check Syntax
Execute
Break
;;count-s: SK-exp -> Number
;; return the number of S in the exp
(define (count-s ask)
  (cond
    [(empty? ask) 0]
    [(cons? (first ask)) (+ (count-s (first ask))
                           (count-s (rest ask)))]
    [(symbol=? (first ask) 'k)
     (count-s (rest ask))]
    [(symbol=? (first ask) 's)
     (+ 1 (count-s (rest ask)))]))
(count-s empty) "should be" 0
(count-s '(s)) "should be" 1
(count-s '(s k s)) "should be" 2
(count-s '((s s k (s (s s k))) k)) "should be" 5
Language: Beginning Student with List Abbreviations.
0
"should be"
0
1
"should be"
1
2
"should be"
2
5
"should be"
5
>
15:2
GC
43,556,864
Read/Write
not running

S-K Combinators: Mathematica

Back

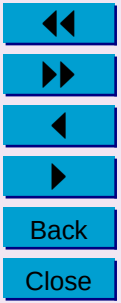
Close

Data Definition

```
Data Definition
An SK-exp is one of:
  {}
  Prepend[SK-exp, "s"]
  Prepend[SK-exp, "k"]
  Prepend[SK-exp, SK-exp]
```

Examples

```
SCount [{}] === 0
SCount[{"S"}] === 1
SCount[{"S", "K"}] === 1
SCount[{"S", {"K"}}] === 1
SCount[{{"S", "S", "K", {"S" , {"S", "S", "K"}}}, "K"}]
  === 5
```



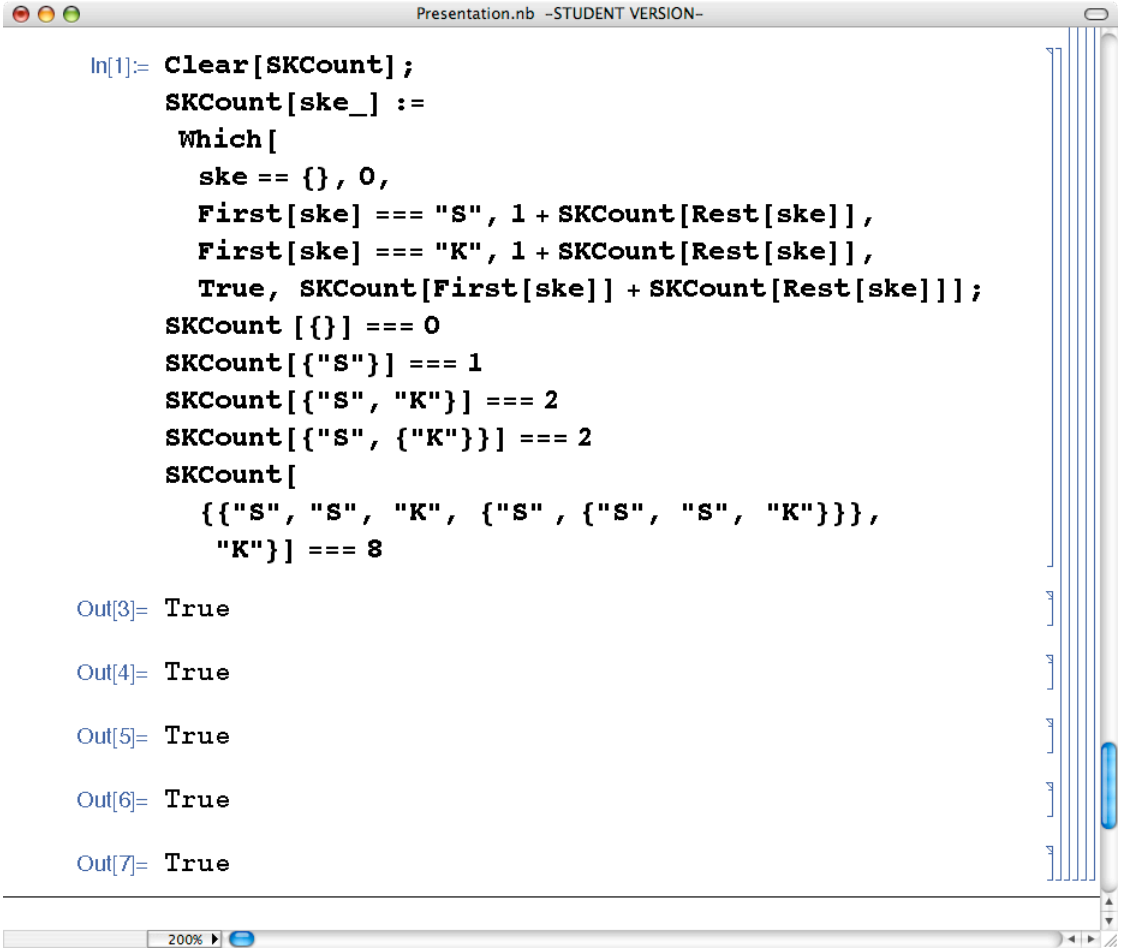
Body

```
Clear[SCount];
SCount[ske_] :=
  Which[
    ske == {}, 0,
    First[ske] === "S", 1 + SCount[Rest[ske]],
    First[ske] === "K", SCount[Rest[ske]],
    True, SCount[First[ske]] + SCount[Rest[ske]]];
```

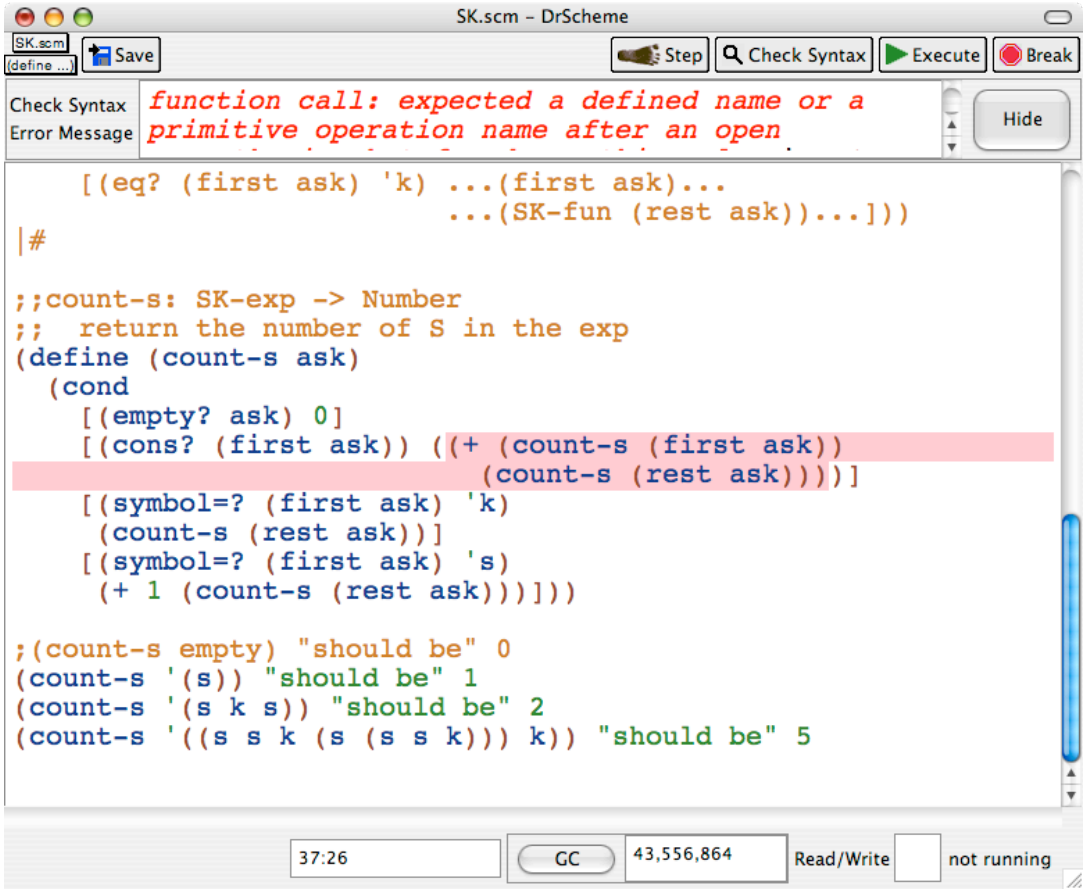
Back

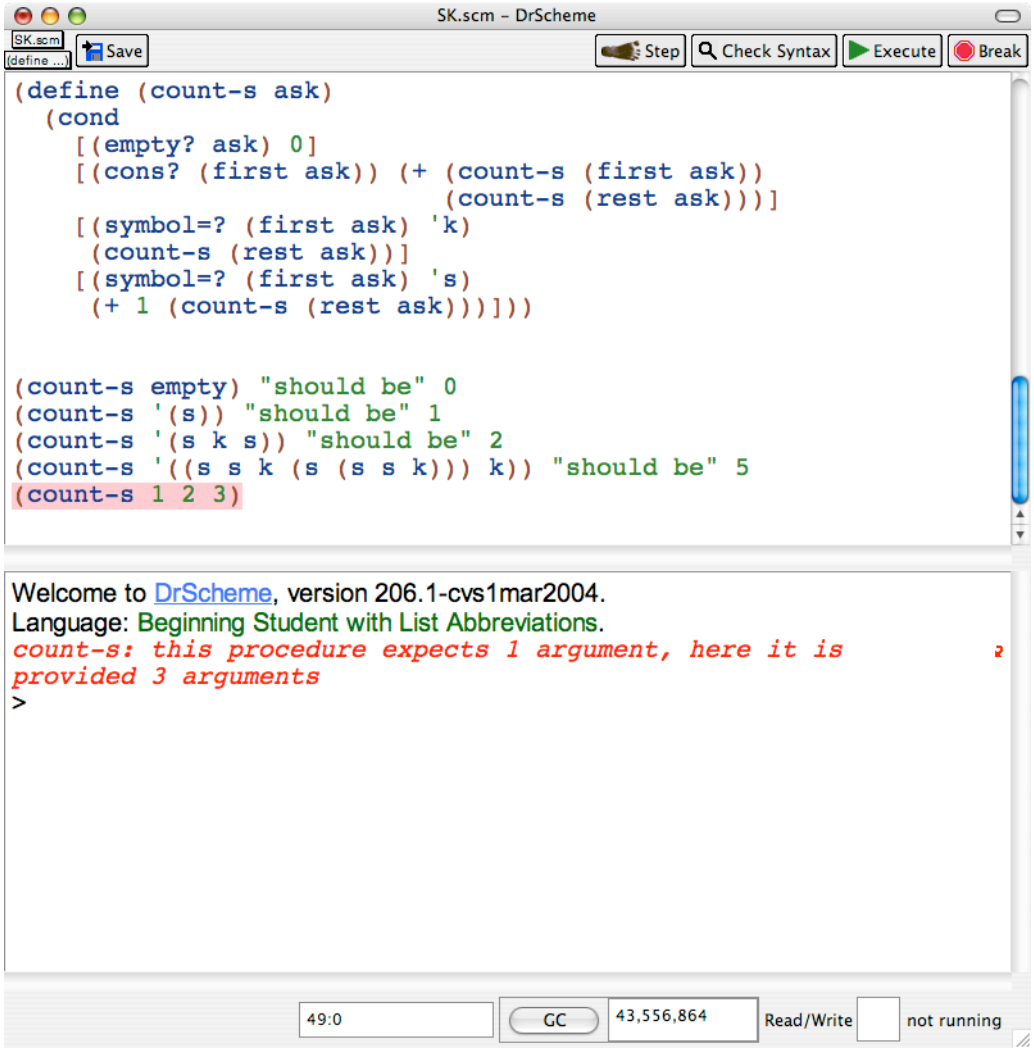
Close

Testing

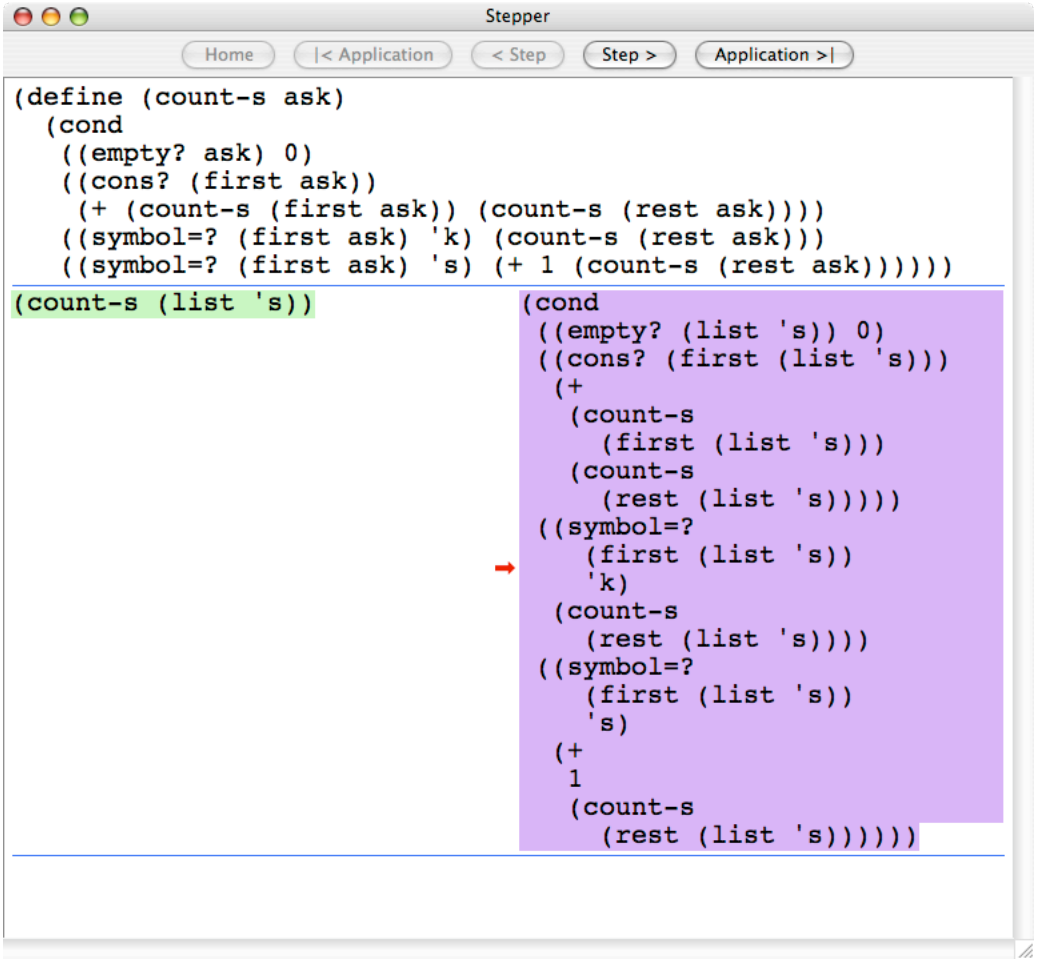

Presentation.nb -STUDENT VERSION-
In[1]:= Clear[SKCount];
SKCount[ske_] :=
Which[
ske == {}, 0,
First[ske] === "S", 1 + SKCount[Rest[ske]],
First[ske] === "K", 1 + SKCount[Rest[ske]],
True, SKCount[First[ske]] + SKCount[Rest[ske]]];
SKCount [{}] === 0
SKCount[{"S"}] === 1
SKCount[{"S", "K"}] === 2
SKCount[{"S", {"K"}}] === 2
SKCount[
{{"S", "S", "K", {"S" , {"S", "S", "K"}}},
"K"}] === 8
Out[3]= True
Out[4]= True
Out[5]= True
Out[6]= True
Out[7]= True
200%

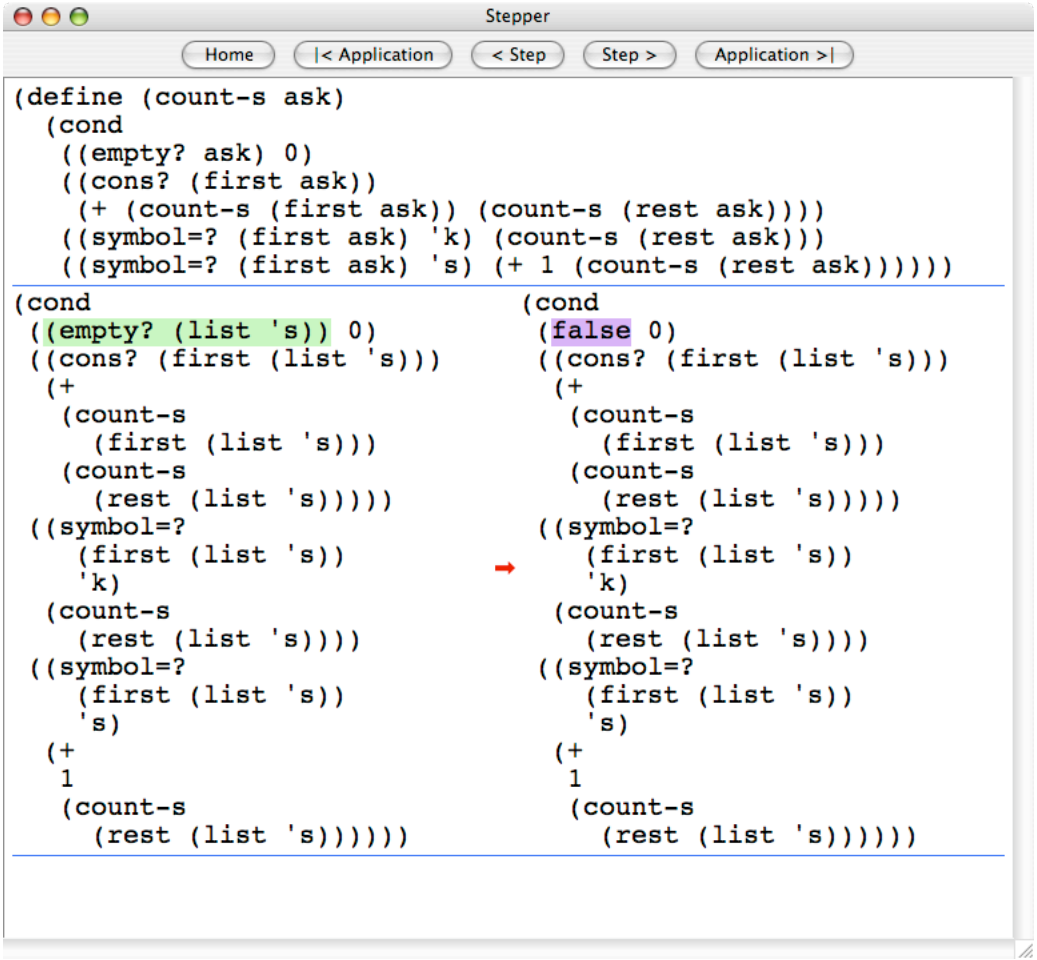
Errors


SK.scm – DrScheme
SK.scm
(define ...)
Save
Step
Check Syntax
Execute
Break
Check Syntax Error Message
function call: expected a defined name or a primitive operation name after an open
Hide
[(eq? (first ask) 'k) ...(first ask)...
...(SK-fun (rest ask))...]))
|#
;;count-s: SK-exp -> Number
;; return the number of S in the exp
(define (count-s ask)
(cond
[(empty? ask) 0]
[(cons? (first ask)) ((+ (count-s (first ask))
(count-s (rest ask))))]
[(symbol=? (first ask) 'k)
(count-s (rest ask))]
[(symbol=? (first ask) 's)
(+ 1 (count-s (rest ask)))]))
;(count-s empty) "should be" 0
(count-s '(s)) "should be" 1
(count-s '(s k s)) "should be" 2
(count-s '((s s k (s (s s k))) k)) "should be" 5
37:26
GC
43,556,864
Read/Write
not running


SK.scm – DrScheme
SK.scm
define ...
Save
Step
Check Syntax
Execute
Break
(define (count-s ask)
(cond
[(empty? ask) 0]
[(cons? (first ask)) (+ (count-s (first ask))
(count-s (rest ask)))]
[(symbol=? (first ask) 'k)
(count-s (rest ask))]
[(symbol=? (first ask) 's)
(+ 1 (count-s (rest ask)))]))
(count-s empty) "should be" 0
(count-s '(s)) "should be" 1
(count-s '(s k s)) "should be" 2
(count-s '((s s k (s (s s k))) k)) "should be" 5
(count-s 1 2 3)
Welcome to DrScheme, version 206.1-cvs1mar2004.
Language: Beginning Student with List Abbreviations.
count-s: this procedure expects 1 argument, here it is
provided 3 arguments
>
49:0
GC
43,556,864
Read/Write
not running

Stepper


Stepper
Home
|< Application
< Step
Step >
Application >|
(define (count-s ask)
(cond
((empty? ask) 0)
((cons? (first ask))
(+ (count-s (first ask)) (count-s (rest ask))))
((symbol=? (first ask) 'k) (count-s (rest ask)))
((symbol=? (first ask) 's) (+ 1 (count-s (rest ask))))))
(count-s (list 's))
(cond
((empty? (list 's)) 0)
((cons? (first (list 's)))
(+
(count-s
(first (list 's)))
(count-s
(rest (list 's)))))
((symbol=?
(first (list 's))
'k)
(count-s
(rest (list 's))))
((symbol=?
(first (list 's))
's)
(+
1
(count-s
(rest (list 's))))))


Stepper
Home
|< Application
< Step
Step >
Application >|
(define (count-s ask)
(cond
((empty? ask) 0)
((cons? (first ask))
(+ (count-s (first ask)) (count-s (rest ask))))
((symbol=? (first ask) 'k) (count-s (rest ask)))
((symbol=? (first ask) 's) (+ 1 (count-s (rest ask))))))
(cond
((empty? (list 's)) 0)
((cons? (first (list 's)))
(+
(count-s
(first (list 's)))
(count-s
(rest (list 's)))))
((symbol=?
(first (list 's))
'k)
(count-s
(rest (list 's))))
((symbol=?
(first (list 's))
's)
(+
1
(count-s
(rest (list 's))))))
(cond
(false 0)
((cons? (first (list 's)))
(+
(count-s
(first (list 's)))
(count-s
(rest (list 's)))))
((symbol=?
(first (list 's))
'k)
(count-s
(rest (list 's))))
((symbol=?
(first (list 's))
's)
(+
1
(count-s
(rest (list 's))))))

Concluding Thoughts

- Programming and NKS
- Little languages
- Future directions

http://www.ccs.neu.edu/home/raymond/TeachNKS