Have FUN
with
Functional Programming

Geert Janssen

Contents Overview

- Theory
  - Lambda expressions
  - Reduction
  - Normal Form
  - Combinators
- Practice:
  - Compilation
  - Representation
  - Graph Reduction
  - Extended Syntax

Introduction

Features of Functional Programming:

- A program is an expression;
  execution evaluates it to a value.
- No side-effects, no assignments
- No pointers, no goto's
- In effect: no statements; sequencing is achieved by function applications
- Functions are first-class objects
- High-order functions
- Strongly-typed, but typing is optional

Clear Contrast

When to evaluate function arguments?

- **Imperative language:**
  Before function body is executed:

  Call by value $\Rightarrow$ eager evaluation, applicative order reduction

- **Functional language:**
  During execution of function body:

  Call by need (call on demand) $\Rightarrow$ lazy evaluation, normal order reduction

What's on the Market

Lisp	McCarthy	1960
FP	Backus	1978
Hope	Burstall	1980
Ponder	Fairbarn	1982
ML	Cardelli	1983
Lazy ML	Johnsson	1984
Miranda	David Turner	1985
Orwell	Philip Wadler	1990
Haskell	Paul Hudak et.al.	1991
Gofer	Mark P. Jones	1991

Lambda Expressions

- Abstract Syntax:

```
λ-Expression ::= Constant
               |  Var
               |  Application
               |  Abstraction

Application ::= λ-Expression λ-Expression

Abstraction ::= λ Var . λ-Expression
```

- Concrete Syntax:

  Application is left-associative and binds stronger than abstraction, which is right-associative.

  We also also introduce $\lambda x_1 x_2 x_3. E$ as short-hand for $\lambda x_1. \lambda x_2. \lambda x_3. E$.

Free & Bound Occurrences

x in expression	occurs free?
k	No
x	Yes
y	No
E1 E2	x free in E1 or x free in E2
λx. E	No
λy. E	x free in E

x in expression	occurs bound?
k	No
x	No
y	No
E1 E2	x bound in E1 or x bound in E2
λx. E	x free in E
λy. E	x bound in E

Conversions/Reductions

α: Consistent name-changing.

$\lambda x. E \leftrightarrow \lambda y. E[y/x]$, y **not free in** E

β: Function application.

$(\lambda x. E) M \leftrightarrow E[M/x]$

η: Abstraction elimination.

$\lambda x. Ex \leftrightarrow E$, x **not free in** E

Recursion

```
FAC = λ n . IF (= n 0) 1 (* n (FAC (- n 1)))

FAC = λ n . (... FAC ...)

FAC = (λ fac . λ n . (... fac ...)) FAC

FAC = H FAC
```

FAC is a fixpoint of H.

Invent function Y that delivers fixpoint of its argument, thus:

```
H (Y H) = Y H
```

then

```
FAC = Y H
```

Does Y exist as lambda expression? Yes:

```
Y = λ h . (λ x . h (x x)) λ x . h (x x)
```

Proof:

```
Y H
= (λ h . (λ x . h (x x)) λ x . h (x x)) H
= (λ x . H (x x)) λ x . H (x x)
= H ((λ x . H (x x)) λ x . H (x x))
= H (Y H)
```

Reduction Order - Normal Form

- A functional program is executed by repeatedly selecting and evaluating the next reducible expression, so-called redex.
- If an expression contains no redexes, evaluation is complete and the expression is said to be in normal form.
- Not every expression has a normal form!
- The 'next' redex need not be unique: reduction may proceed along different sub-expression orders.
- Some reduction orders may reach a normal form, others might not!
- Do different reduction orders always lead to the same normal form?

Need some theory here

Church-Rosser Theorems

Theorem 1: If $E1 \leftrightarrow E2$ then there exists an expression E such that $E1 \rightarrow E$ and $E2 \rightarrow E$.

Corollary No expression can be converted to two distinct normal forms; reductions that terminate reach the same result.

Theorem 2: If $E1 \rightarrow E2$ and E2 is in normal form, then there exists a normal order reduction sequence from E1 to E2.

Normal order reduction:
reduce leftmost outermost redex first

Bottom line: If an expression has a normal form then normal order reduction will surely lead to it. Any other reduction order will never give a wrong result; it simply might not terminate.

Normal Order Reduction

Guarantees that:

Arguments to functions are evaluated only when needed.

How to ensure that:

When evaluated, arguments are evaluated only once?

Answer: by so-called graph-reduction.

Combinators - SKI-ing

- **Definition**: a combinator is a closed lambda-expression, i.e., it has no free variables (it is a pure function).
- For example, Y is a combinator.
- Define:

  $$\mathbf{S} = \lambda\ \mathbf{f}\ .\ \lambda\ \mathbf{g}\ .\ \lambda\ \mathbf{x}\ .\ \mathbf{f}\ \mathbf{x}\ (\mathbf{g}\ \mathbf{x})$$

  $$\mathbf{K} = \lambda\ \mathbf{x}\ .\ \lambda\ \mathbf{y}\ .\ \mathbf{x}$$

  $$\mathbf{I} = \lambda\ \mathbf{x}\ .\ \mathbf{x}$$

- (Syntactic) Transformations:

$$\lambda\ \mathbf{x}\ .\ \mathbf{E1}\ \mathbf{E2} \quad \Rightarrow \quad \mathbf{S}\ (\lambda\ \mathbf{x}\ .\ \mathbf{E1})\ (\lambda\ \mathbf{x}\ .\ \mathbf{E2})$$

$$\lambda\ \mathbf{x}\ .\ \mathbf{c} \quad \Rightarrow \quad \mathbf{K}\ \mathbf{c}\ (\textbf{Note: } \mathbf{c} \neq \mathbf{x})$$

$$\lambda\ \mathbf{x}\ .\ \mathbf{x} \quad \Rightarrow \quad \mathbf{I}$$

SKI-Compilation

```
SK-Expression Compile (λ-Expression E)
{
  while (E contains abstraction) {
    Choose any innermost abstraction A;

    switch (A) {
    case λ x . E1 E2:
      replace A by S (λ x . E1) (λ x . E2) in E;
      break;

    case λ x . c:
      replace A by K c in E;
      break;

    case λ x . x:
      replace A by I in E;
      break;
    }
  }
  return E;
}
```

SK-Expression

```
SK-Expression ::= Constant
               |  Combinator
               |  FreeVar
               |  Application

Combinator ::= S | K | I

Application ::= SK-Expression SK-Expression
```

- Ergo: no more abstractions!
  No longer need to worry about free and bound variables.

Example

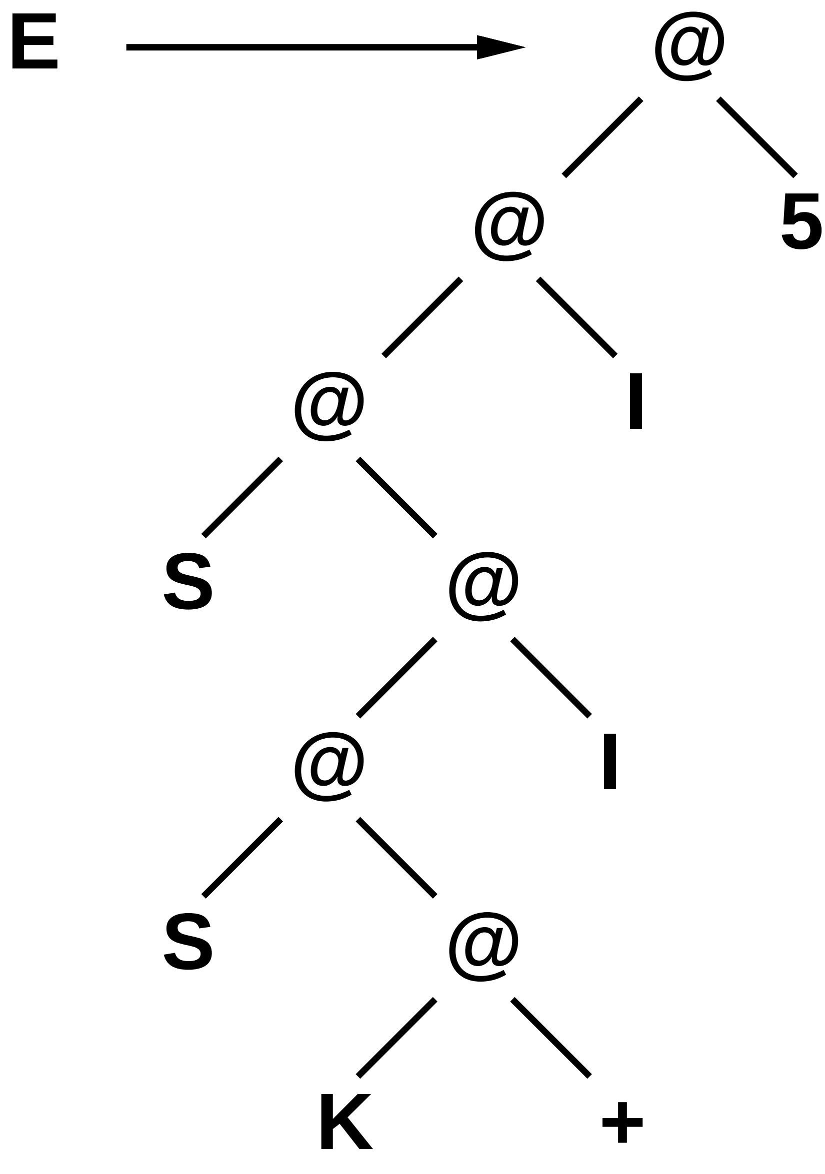
- Compilation:

$$
\begin{array}{ll}
 & \texttt{(}\lambda\texttt{ x . + x x) 5} \\
\Rightarrow & \texttt{S (}\lambda\texttt{ x . + x) (}\lambda\texttt{ x . x) 5} \\
\Rightarrow & \texttt{S (S (}\lambda\texttt{ x . +) (}\lambda\texttt{ x . x)) (}\lambda\texttt{ x . x) 5} \\
\Rightarrow & \texttt{S (S (K +) (}\lambda\texttt{ x . x)) (}\lambda\texttt{ x . x) 5} \\
\Rightarrow & \texttt{S (S (K +) I) (}\lambda\texttt{ x . x) 5} \\
\Rightarrow & \texttt{S (S (K +) I) I 5}
\end{array}
$$

- Verify by reduction:

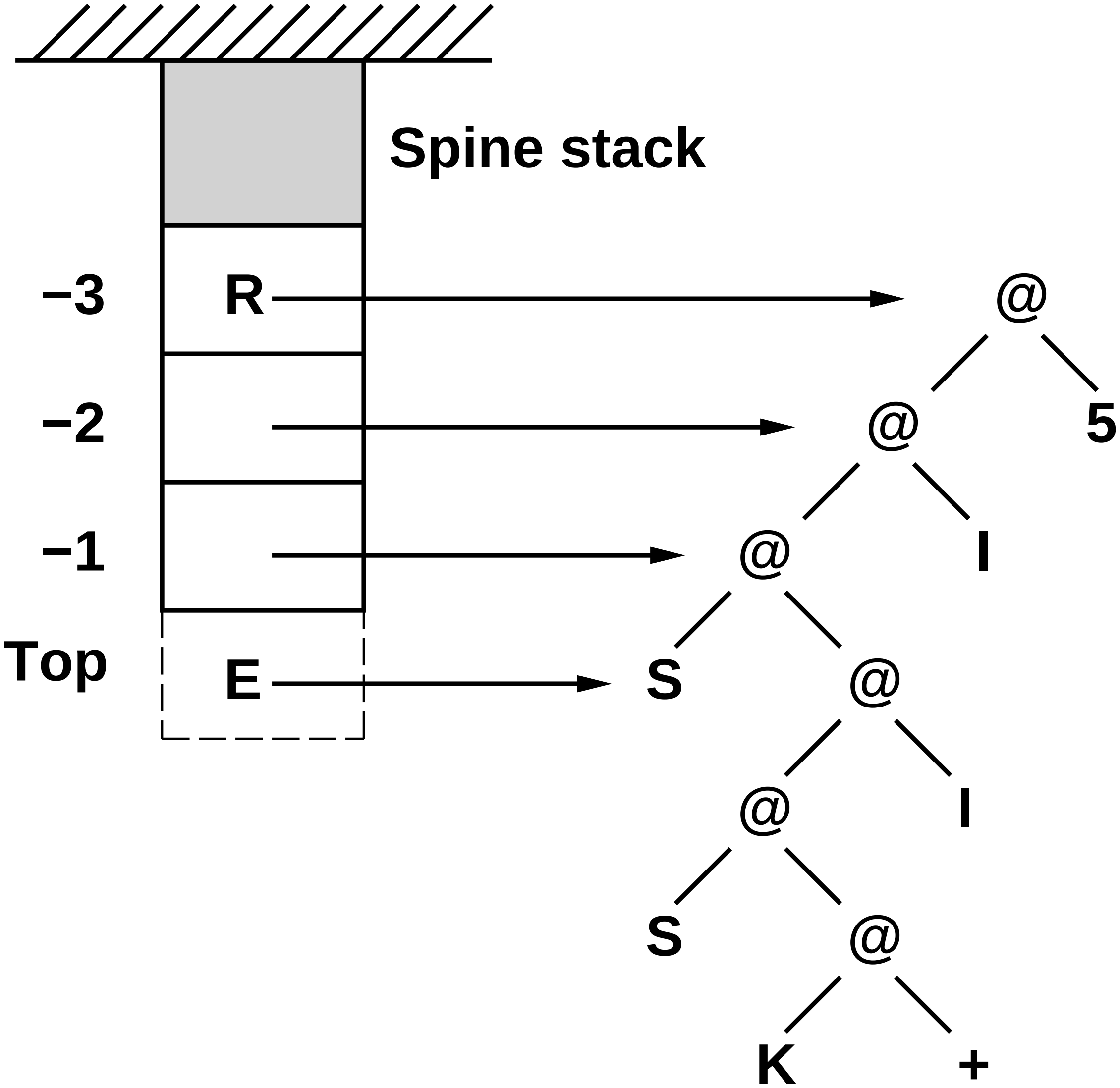
$$
\begin{array}{ll}
 & \texttt{S (S (K +) I) I 5} \\
\rightarrow & \texttt{S (K +) I 5 (I 5)} \\
\rightarrow & \texttt{K + 5 (I 5) (I 5)} \\
\rightarrow & \texttt{+ (I 5) (I 5)} \\
\rightarrow & \texttt{+ 5 (I 5)} \\
\rightarrow & \texttt{+ 5 5} \\
\rightarrow & \texttt{10}
\end{array}
$$

Representation of SK-Expression



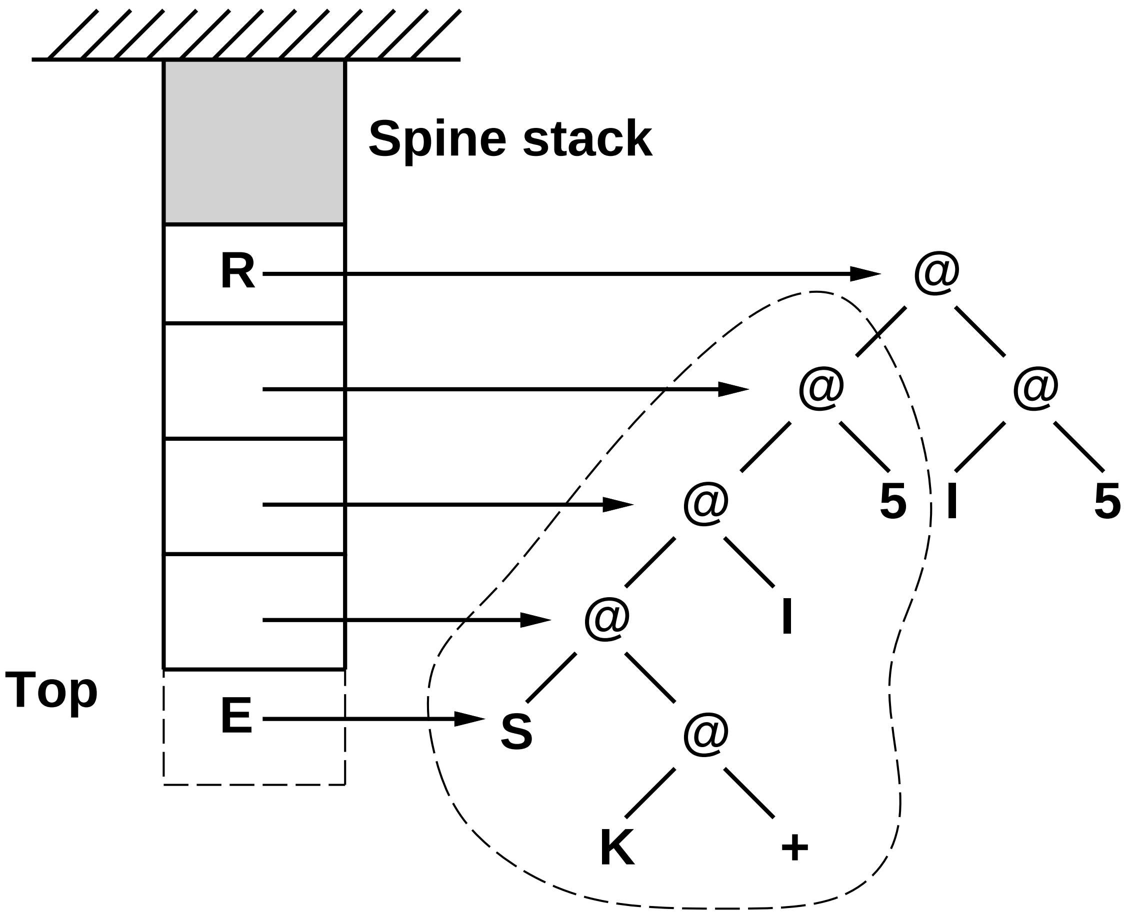
S (S (K +) I) I 5

Graph Reduction 1



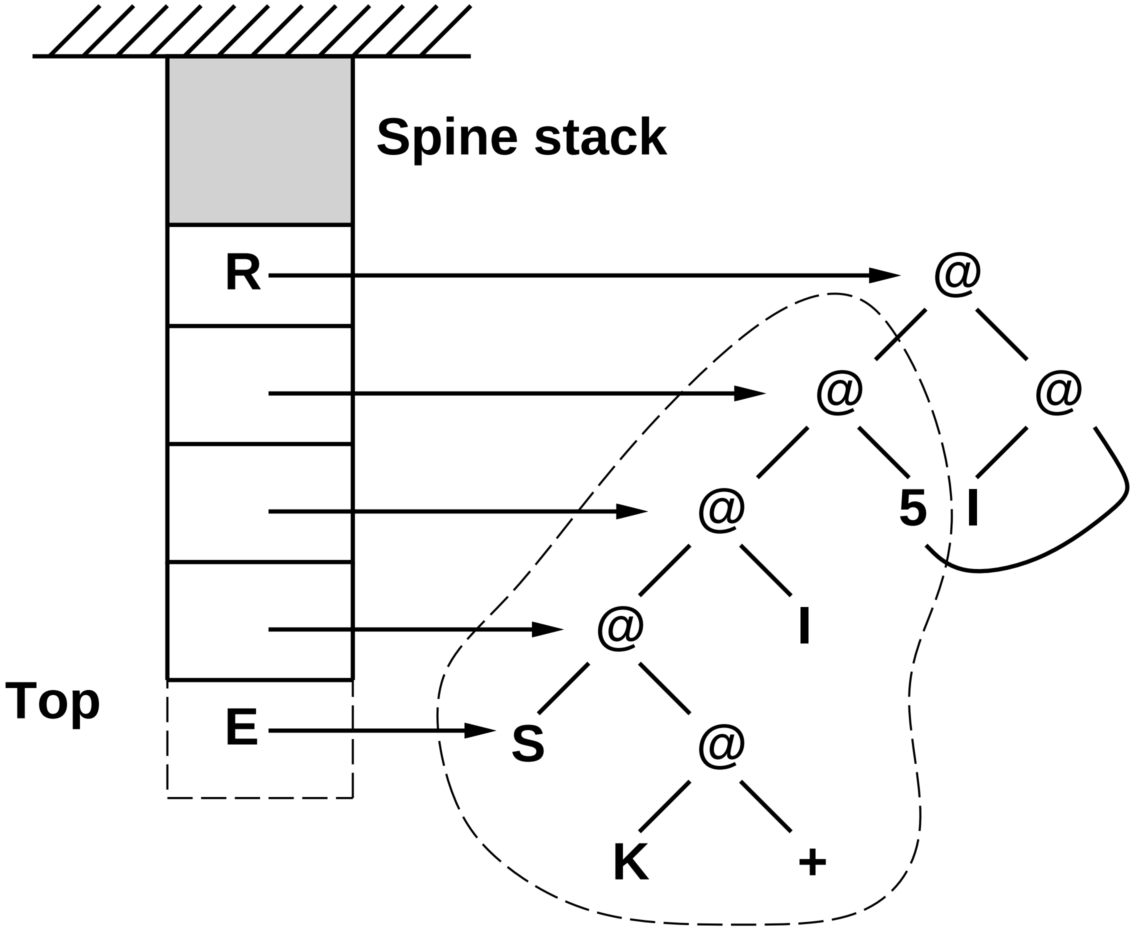
S (S (K +) I) I 5

Graph Reduction 2



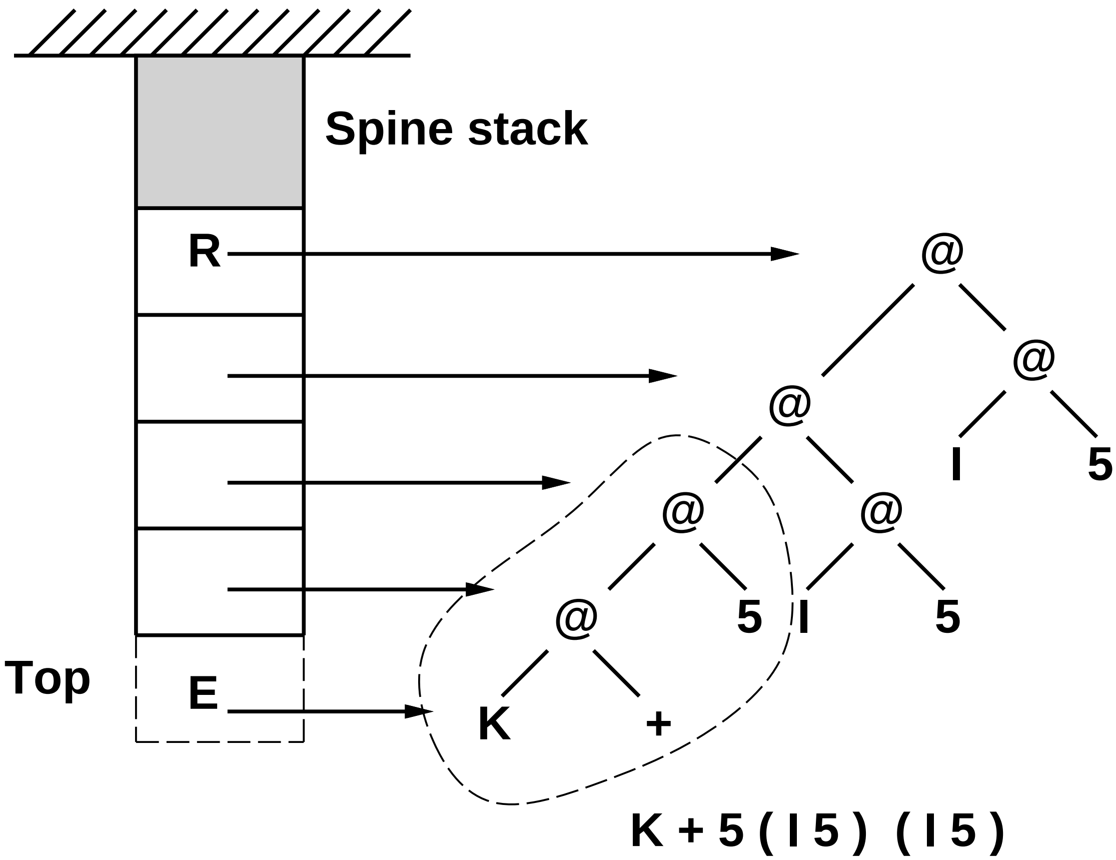
S (K +) I 5 (I 5)

Graph Reduction 2b

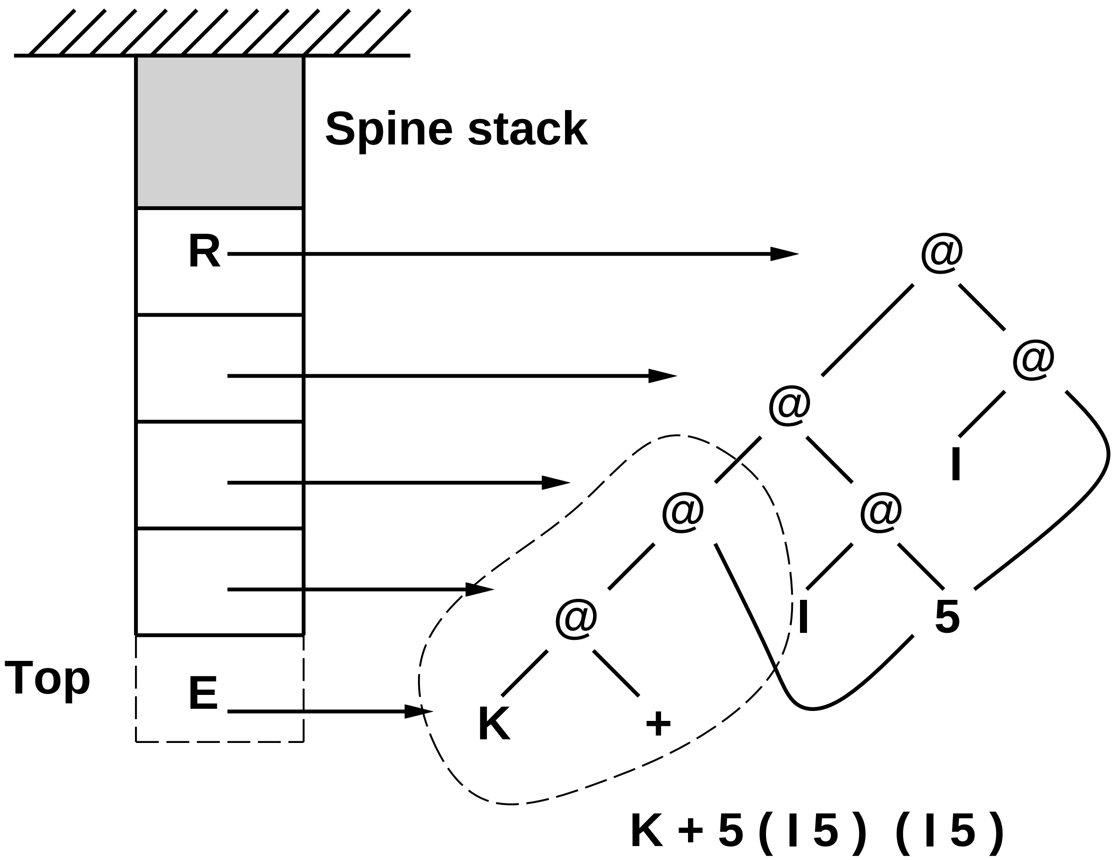


S (K +) I 5 (I 5)

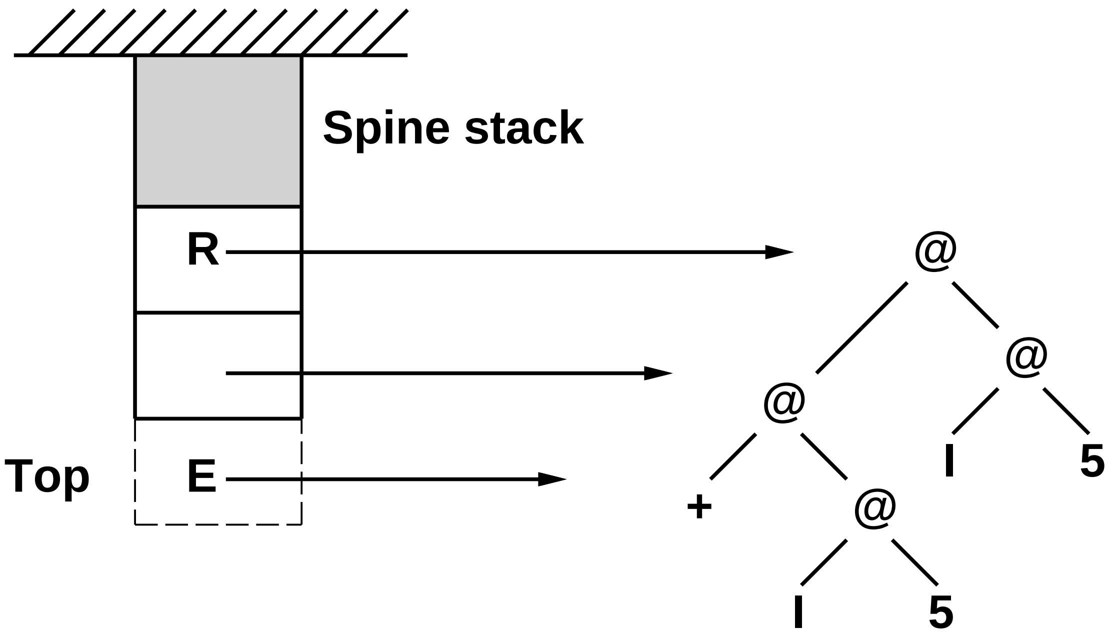
Graph Reduction 3



Graph Reduction 3b

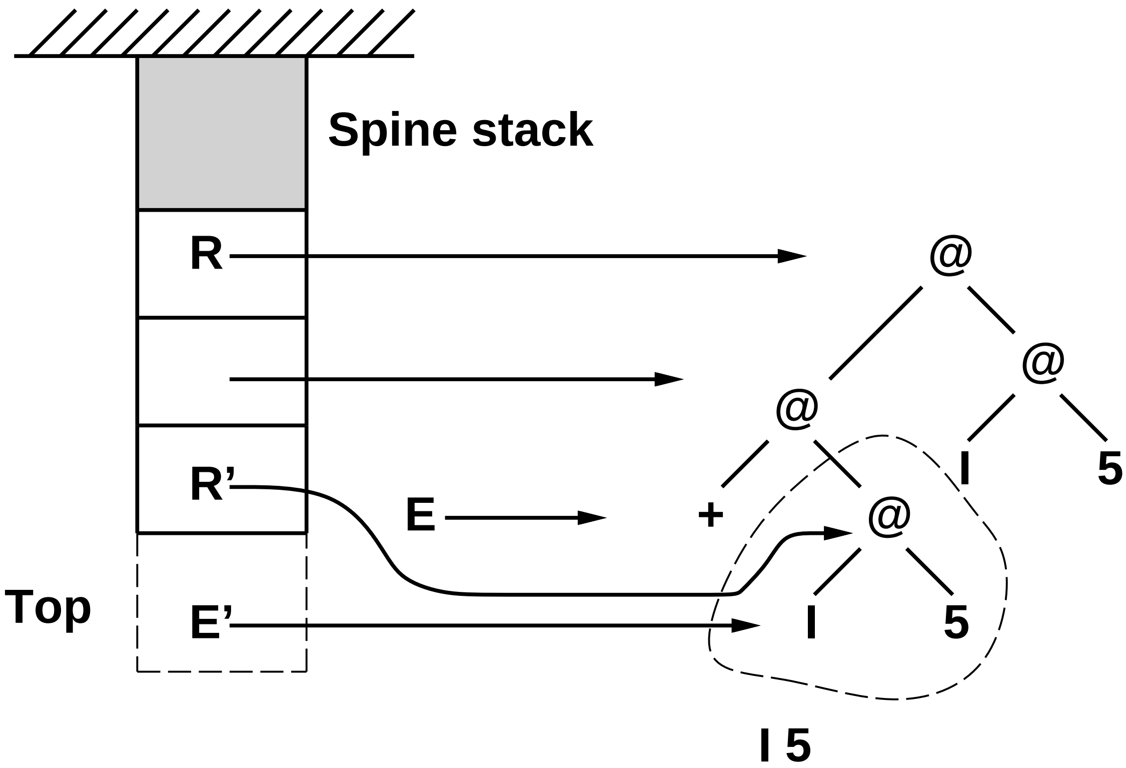


Graph Reduction 4

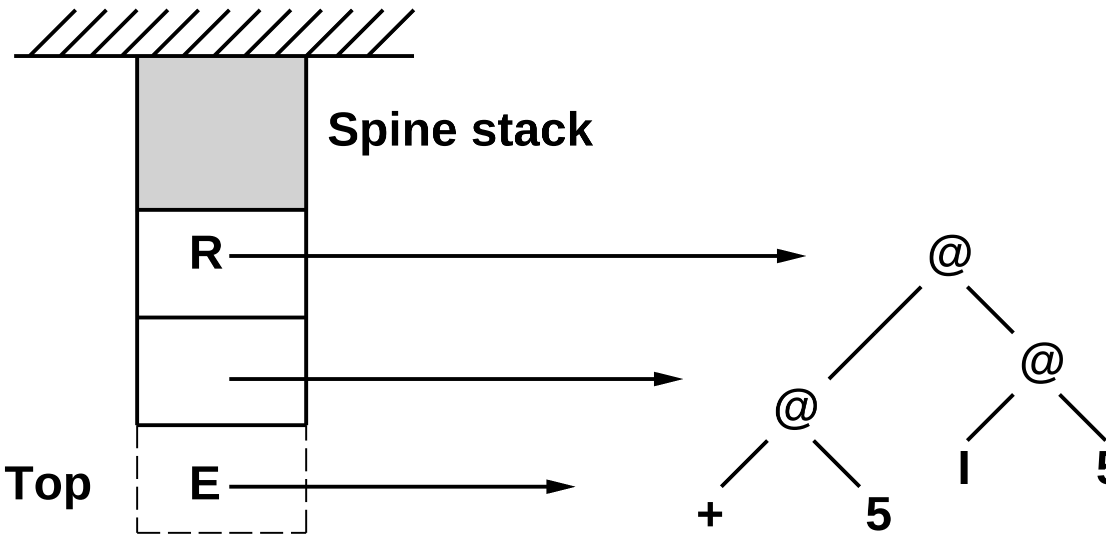


+ (I 5) (I 5)

Graph Reduction 5

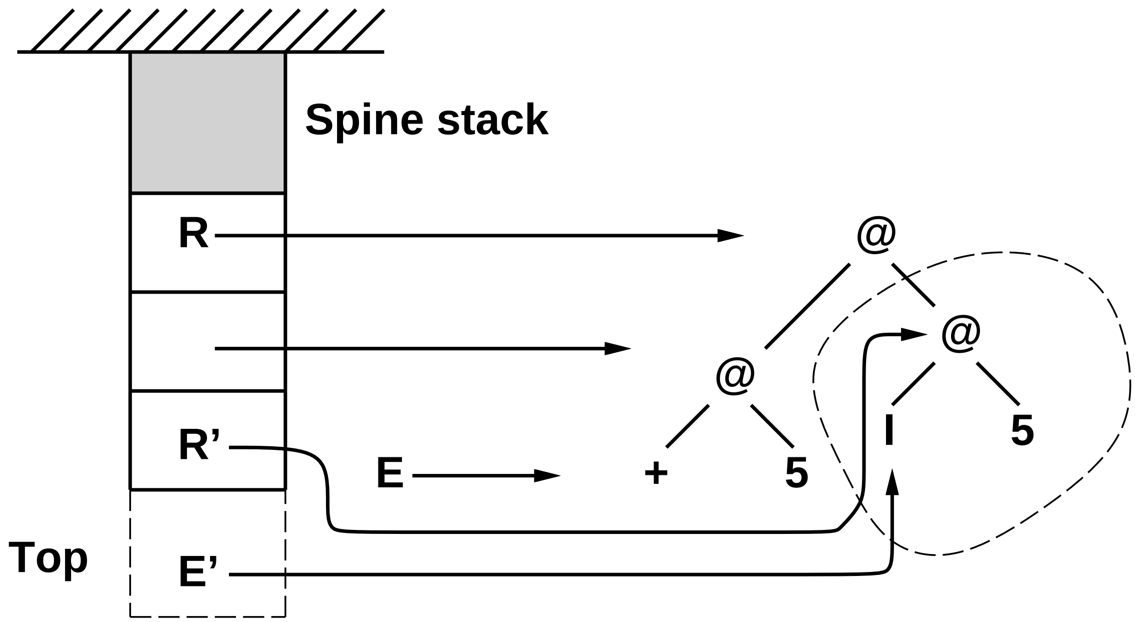


Graph Reduction 6



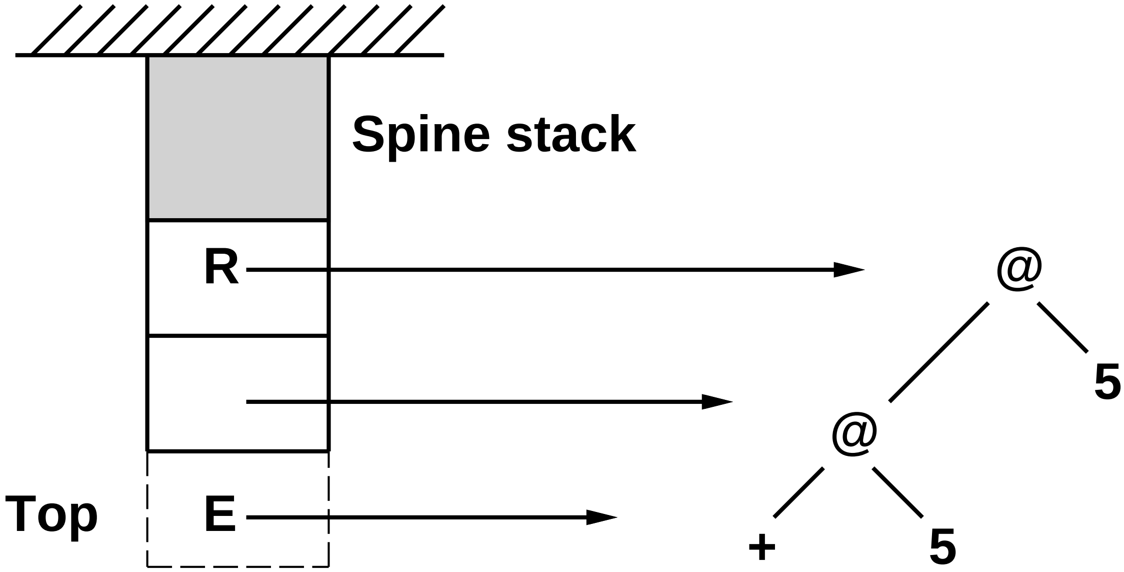
+ 5 (I 5)

Graph Reduction 7



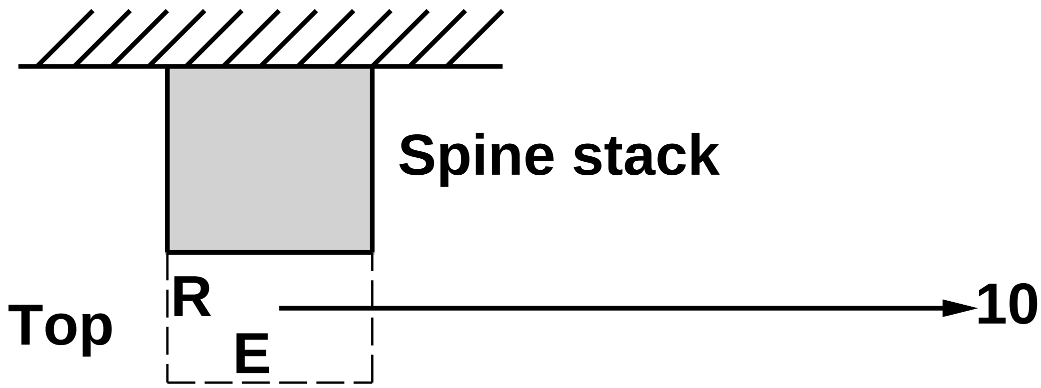
I 5

Graph Reduction 8



+ 5 5

Graph Reduction 9



Interpreter = Reductor

```
SK-expression Reduce (SK-expression E)
{
  saveTop:=Top;
  R:=E;
  forever {
    switch (E) {
/* COMBINATOR: */
    case I:     /* Redex: I f */
      *(Top-1):=f;                  drop(1); break;
    case K:     /* Redex: K c x */
      *(Top-2):=c;                  drop(2); break;
    case S:     /* Redex: S f g x */
      *(Top-3):=f x (g x);          drop(3); break;
    case Y:     /* Redex: Y f */
      *(Top-1):=f (Y f);            drop(1); break;
/* APPLICATION: */
    case @:     /* Redex: E1 E2 */
      push(E); E:=E1;
      break;
/* BUILT-IN FUNCTIONS: */
    case +:     /* Redex: + E1 E2 */
      *(Top-2):=Reduce(E1) + Reduce(E2);
      drop(2);
      break;
/* ANYTHING ELSE, I.E., CONSTANTS & FREE VARS: */
    default:
      Top:=saveTop;
      return R;
    }
  }
}
```

Extended Syntax

let { v_i **=** Bi **}+ in** E
$\Rightarrow$ **let** v_1 **=** B1 **in (let** v_2 **=** B2 **in ... in** E**)**

let v **=** B **in** E
$\Rightarrow$ **(**λ v **.** E**)** B

letrec v **=** B **in** E
$\Rightarrow$ **let** v **= Y (**λ v **.** B**) in** E

letrec { v_i **=** Bi **}+ in** E
$\Rightarrow$ **(UNPACK-n (**λ v_i**’s .** E**))**
(Y (UNPACK-n (λ v_i**’s . [** Bi **]))).**

E **where {** v_i **=** Bi **}+ end**
$\Rightarrow$ **letrec {** v_i **=** Bi **}+ in** E

More Goodies

- Lists, Arithmetic Series
- List comprehensions
- Pattern matching
- User-defined types

Lists, Arithmetic Series

- Syntax:

```
List ::= '[' ']'
      |  '[' ':' ']'
      |  '[' λ-Expr ':' ']'
      |  '[' { λ-Expr / ',' }+ ']'
      |  '[' { λ-Expr / ',' }+ ':' λ-Expr ']'

      |  '[' λ-Expr '..' ']'
      |  '[' λ-Expr '..' λ-Expr ']'
      |  '[' λ-Expr ',' λ-Expr '..' ']'
      |  '[' λ-Expr ',' λ-Expr '..' λ-Expr ']'
```

- Examples:

`[]`	$\rightarrow$	`[]`
`[:] 1 [ 2, 3 ]`	$\rightarrow$	`[ 1, 2, 3 ]`
`[ 1, 2 : [ 3, 4 ]]`	$\rightarrow$	`[ 1, 2, 3, 4 ]`
`[ 1..5 ]`	$\rightarrow$	`[ 1, 2, 3, 4, 5 ]`
`[ 1, 3..8 ]`	$\rightarrow$	`[ 1, 3, 5, 7 ]`
`[ 1.. ]`	$\rightarrow$	`[ 1, 2, 3, 4, ...`

List Comprehensions

- Syntax:

```
ListComprehension ::=
     '[' λ-Expression '|' { Qualifier / ';' }+ ']'

Qualifier ::= Generator | Filter

Generator ::= { Var / ',' }+ '<-' λ-Expression

Filter ::= λ-Expression
```

- Examples:

```
  [ * n n | n <- [ 0.. ] ] →

      [ 0, 1, 4, 9, 16, 25, ...

  [ n | n <- [ 0..8 ]; == (% n 2) 0 ] →

      [ 0, 2, 4, 6, 8 ]

  [ [x,y] | x <- [1..3]; y <- [1..x] ] →

      [ [ 1, 1 ], [ 2, 1 ], [ 2, 2 ],
        [ 3, 1 ], [ 3, 2 ], [ 3, 3 ] ]
```