Compiling Functional Programs

Mooly Sagiv

Chapter 7

http://www.cs.tau.ac.il/~msagiv/courses/wcc10.html

Main features of Haskel

- No side effects
  - Referential Transparency
- List comprehension
- Pattern matching
- Higher order functions
  - Curried notions
- Polymorphic typing
- Lazy evaluation

Factorial in Haskel vs. C

```
fac 0 = 1
fac n = n * fac (n -1)
```

```
int fac(int n) {
    int product = 1;
    while (n > 0) {
        product *= n ;
        n --;
        }
     return product;
}
```

Function Application

- Concise syntax
- No argument parenthesizes
  - f 11 13
- Function application is left associative and has higher precedence than any operator
  - g g n = (g g) n
  - g n + 1 = (g n) + 1

Offside rule

- Layout characters matter to parsing

```
divide x 0 = inf
divide x y = x / y
```

- Everything below and right of = in equations defines a new scope
- Applied recursively

```
fac n = if (n ==0) then 1 else prod n (n-1)
        where
            prod acc n = if (n == 0) then acc
                         else prod (acc * n) (n -1)
```

- Lexical analyzer maintains a stack

Lists

- Part of all functional programs since Lisp
- Empty list [] = Nil
- [1]
- [1, 2, 3, 4]
- [4, 3, 7, 7, 1]
- [“red”, “yellow”, “green”]
- [1 .. 10]
- Can be constructed using “:” infix operator
  - [1, 2, 3] = (1 : (2 : (3 : [])))
  - range n m = if n > m then []
    else (n: range (n+1) m)

List Comprehension

- Inspired by set comprehension
  $S = \{n^2 \mid n \in \{1, \ldots, 100\} \wedge \text{odd } n\}$
- Haskel code
  s = [n^2 | n <- [1..100], odd n]
  "n square such that n is an element of [1..100] and n is odd"
- Qsort in Haskel

```
qsort [] = []
qsort (x: xs) = qsort [y | y <- xs, y < x]
                ++ [x]
                ++ qsort[y | y <- xs, y >= x]
```

Pattern Matching

- Convenient way to define recursive functions
- A simple example

```
fac 0 = 1
fac n = n * fac (n-1)
```

- Equivalent code

```
fac n = if (n == 0) then 1
        else n * fac (n -1)
```

- Another example

```
length [] = 0
length (x: xs) = 1 + length xs
```

- Equivalent code

```
length list = if (list == []) then 0
              else let
                  x = head list
                  xs = tail list
              in 1 + length xs
```

Polymorphic Typing

- Polymorphic expression has many types
- Benefits:
  - Code reuse
  - Guarantee consistency
- The compiler infers that in
  length [] = 0
  length (x: xs) = 1 + length xs
  - length has the type [a] -> int
    length :: [a] -> int
- Example expressions
  - length [1, 2, 3] + length [“red”, “yellow”, “green”]
  - length [1, 2, “green”] // invalid list
- The user can optionally declare types
- Every expression has the most general type
- “boxed” implementations

Referential Transparency

- Expressions in Haskel have no side effects
- Usually requires more space
  add_one [] = []
  add_one (x xs) = x +1 : add_one xs
- Can be tolerated by garbage collection and smart compilers
- Input/Output operations can be also be defined using Monads

Higher Order Functions

- Functions are first class objects
  - Passed as parameters
  - Returned as results

Example Higher Order Function

- The differential operator
  Df = f' where f'(x) = lim $_{h \downarrow o}$ (f(x+h)-f(x))/h
- In Haskel
```
diff f = f_
         where
           f_ x = (f  (x +h) – f x) / h
           h = 0.0001
```
- diff :: (float -> float) -> (float -> float)
- (diff square) 0 = 0.0001
- (diff square) 0.0001 = 0.0003
- (diff (diff square)) 0 = 2

Currying

- Functions can be created by partially applying a function to some of its arguments
- deriv f x = (f (x + h) – f x) / h
  where h = 0.0001
- deriv f x == (diff f) x
- Non semantic difference by Unary and n-ary functions
- $f\ e_1\ e_2 \ldots e_n = (^n((f\ e_1)\ e_2) \ldots e_n)$
- Complicates compilation

Lazy vs. Eager Evaluation

- When to evaluate expressions
- A simple example
  let const c x = c in const 1 (2 + 3)
- In eager evaluation
  const 1 (2 + 3) $\propto$ const 1 5 = 1
- In lazy Evaluation (Haskel)
  const 1 (2 + 3) $\propto$ 1
- Another example
  let const c x = c in const 1 (2 / 0)

Benefits of Lazy Evaluation

- Define streams
  main = take 100 [1 ..]
- deriv f x = lim [(f (x + h) – f x) / h | h <- [1/2^n | n <- [1..]]]
  where lim (a: b: lst) = if abs(a/b -1) < eps then b
  else lim (b: lst)
  eps = 1.0 e-6
- Lower asymptotic complexity
- Language extensibility
  - Domain specific languages
- But some costs

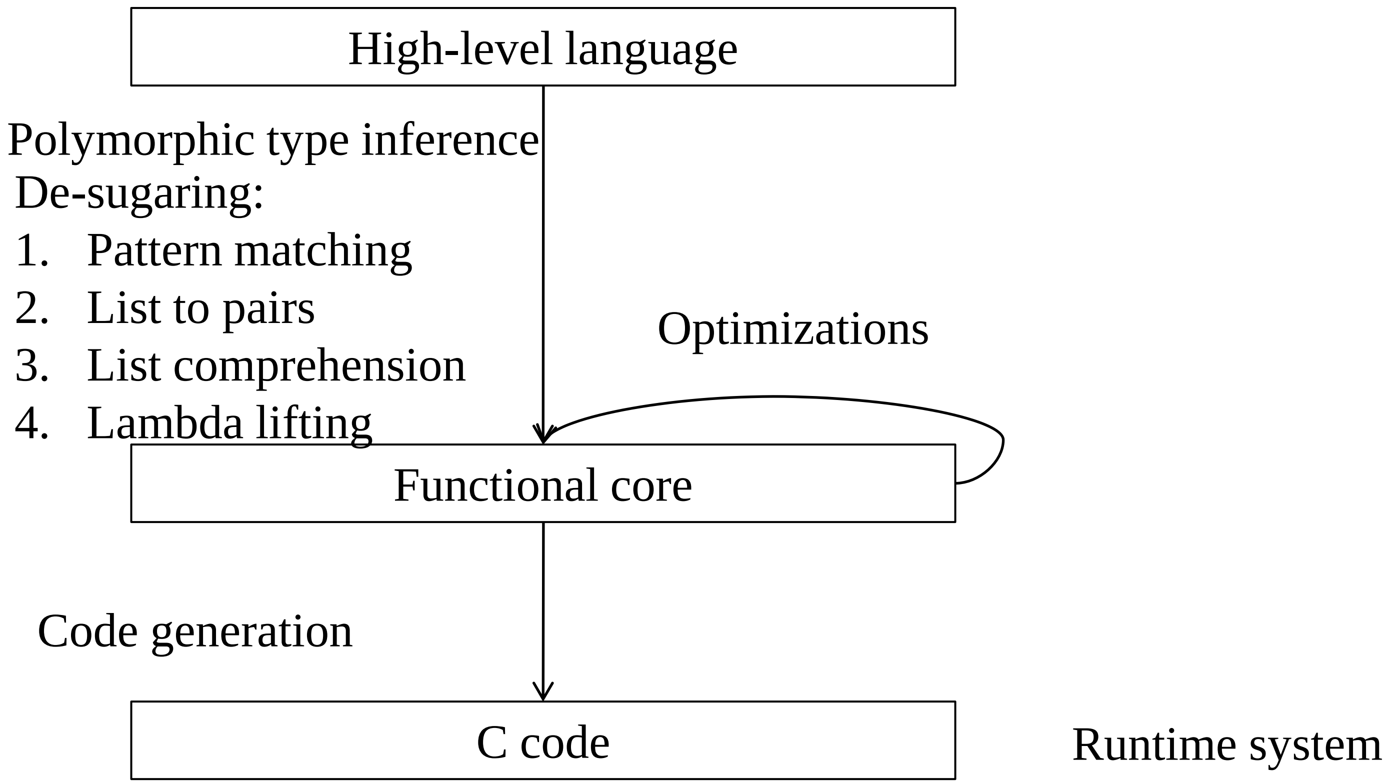
Functional Programming Languages

PL	types	evaluation	Side-effect
scheme	Weakly typed	Eager	yes
ML OCAML F#	Polymorphic strongly typed	Eager	References
Haskel	Polymorphic strongly typed	Lazy	None

Compiling Functional Programs

Compiler Phase	Language Aspect
Lexical Analyzer	Offside rule
Parser	List notation List comprehension Pattern matching
Context Handling	Polymorphic type checking
Run-time system	Referential transparency Higher order functions Lazy evaluation

Structure of a functional compiler

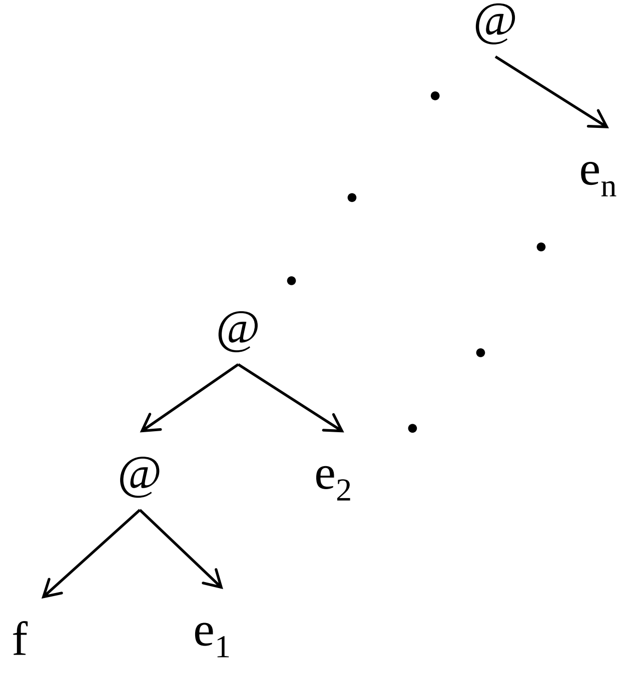


Graph Reduction

- The runtime state of a lazy functional program can be represented as a direct graph
  - Nodes represent arguments in expressions
  - Edges between functions and argument
- An execution is simulated with a graph reduction
- Supports laziness and higher order functions

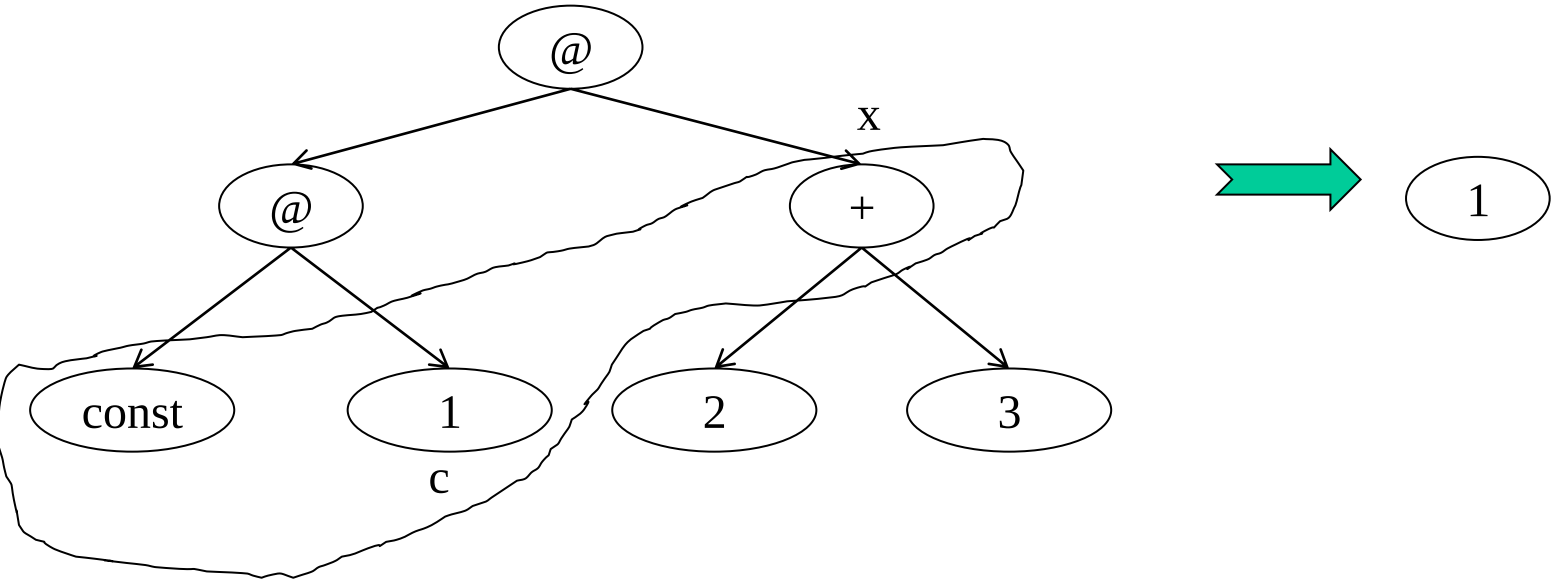
Function Application

- $f\ e_1\ e_2 \ldots e_n = (^n((f\ e_1)\ e_2) \ldots e_n)$
  $= (^n((f\ @\ e_1)\ \ @\ e_2) \ldots e_n)$



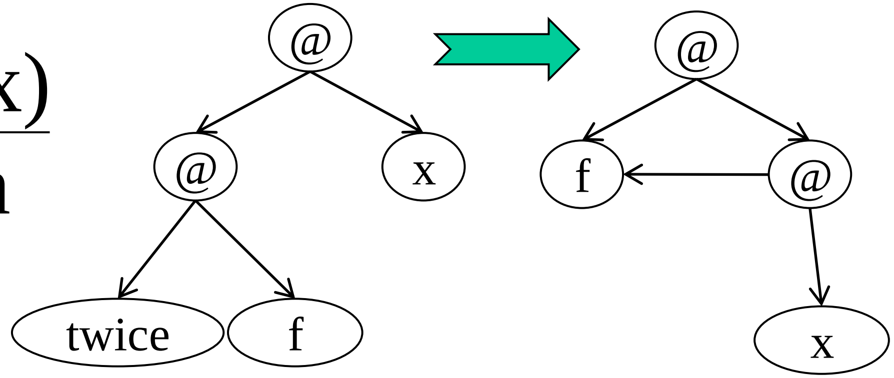
A Simple Example

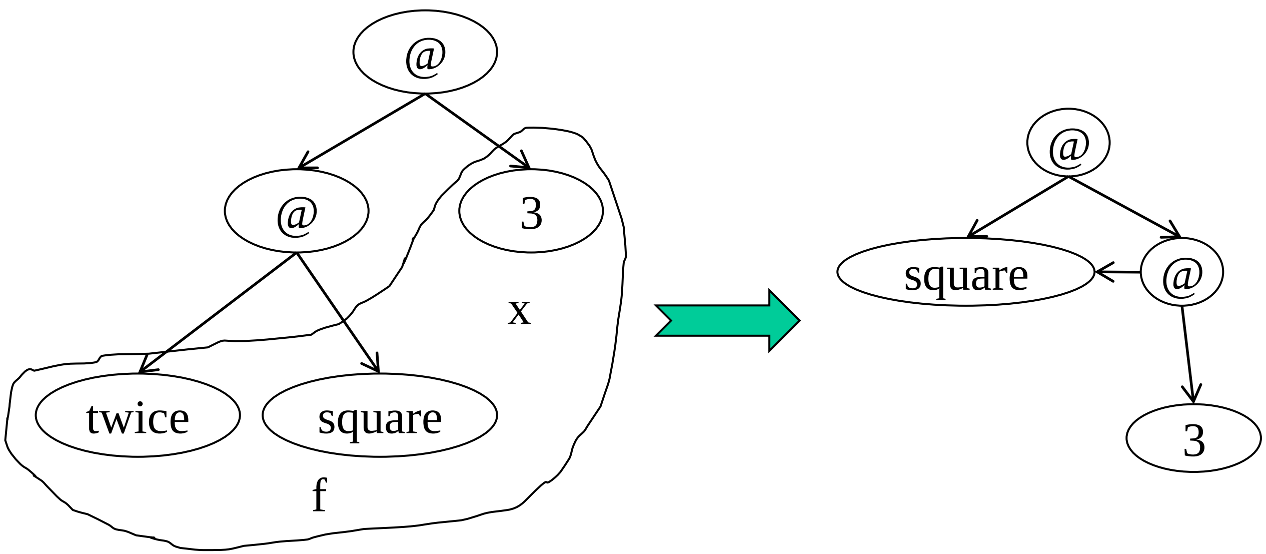
- let const c x = c in const 1 (2 + 3)



Another Example

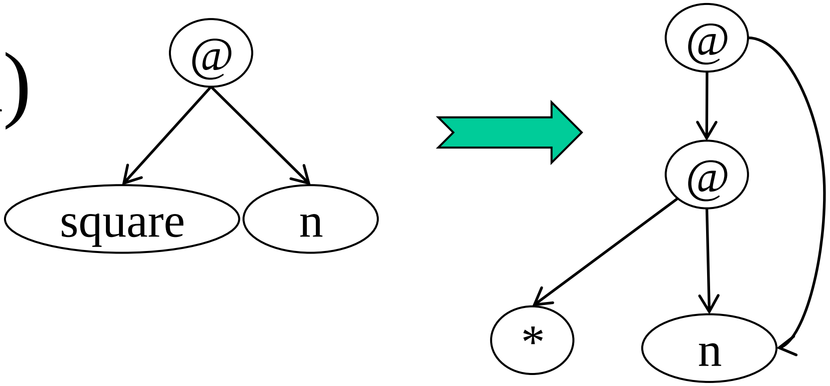
- let twice f x = f (f x)
  square n = n * n
  in
  twice square 3

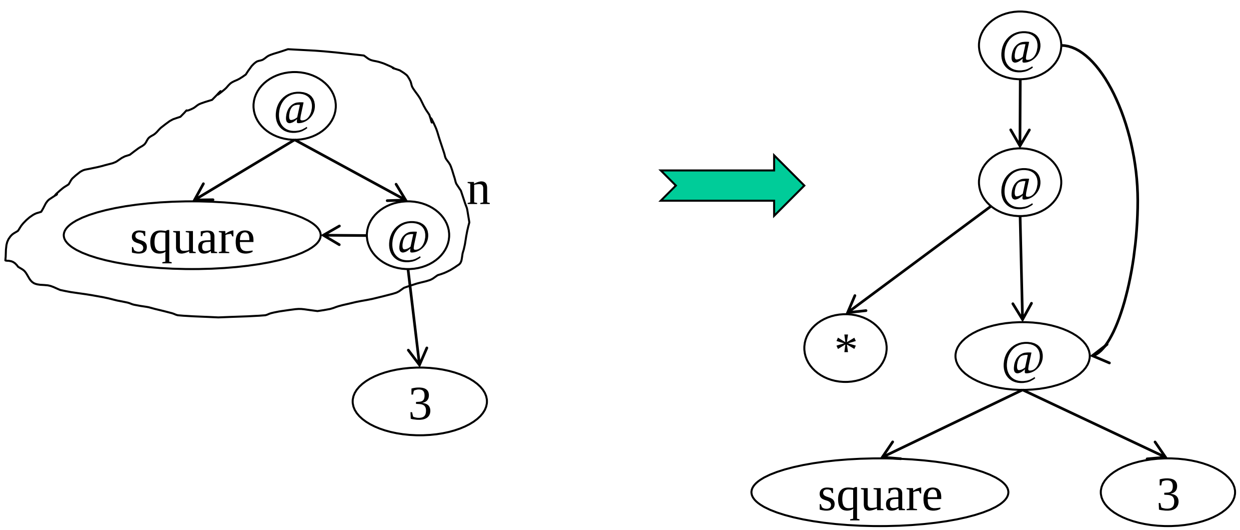




Another Example (cont1)

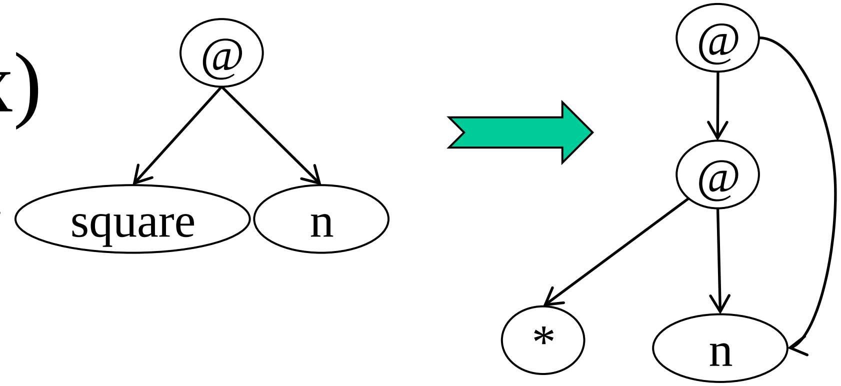
- let twice f x = f (f x)
  square n = n * n
  in
  twice square 3

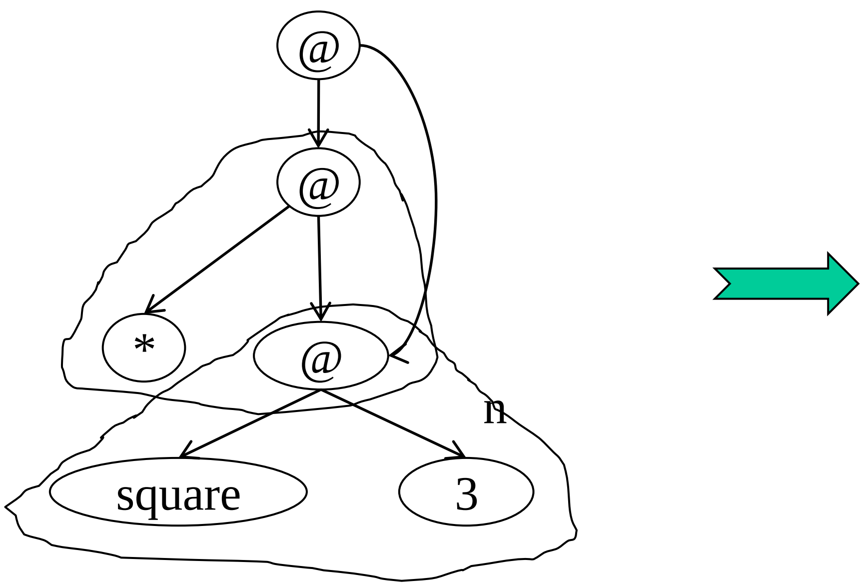


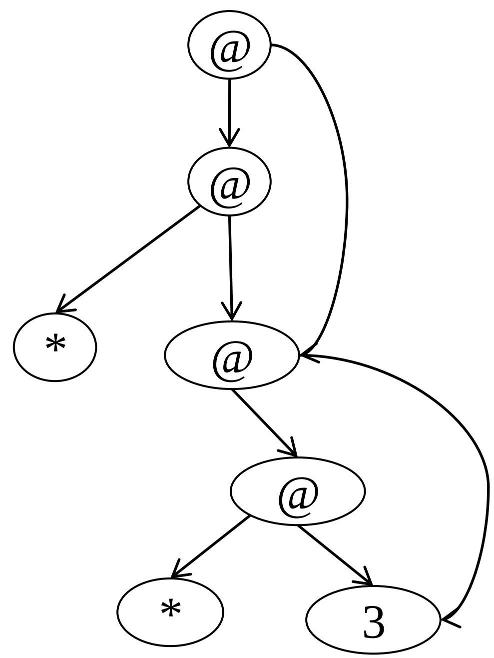


Another Example (cont2)

- let twice f x = f (f x)
  square n = n * n
  in
  twice square 3





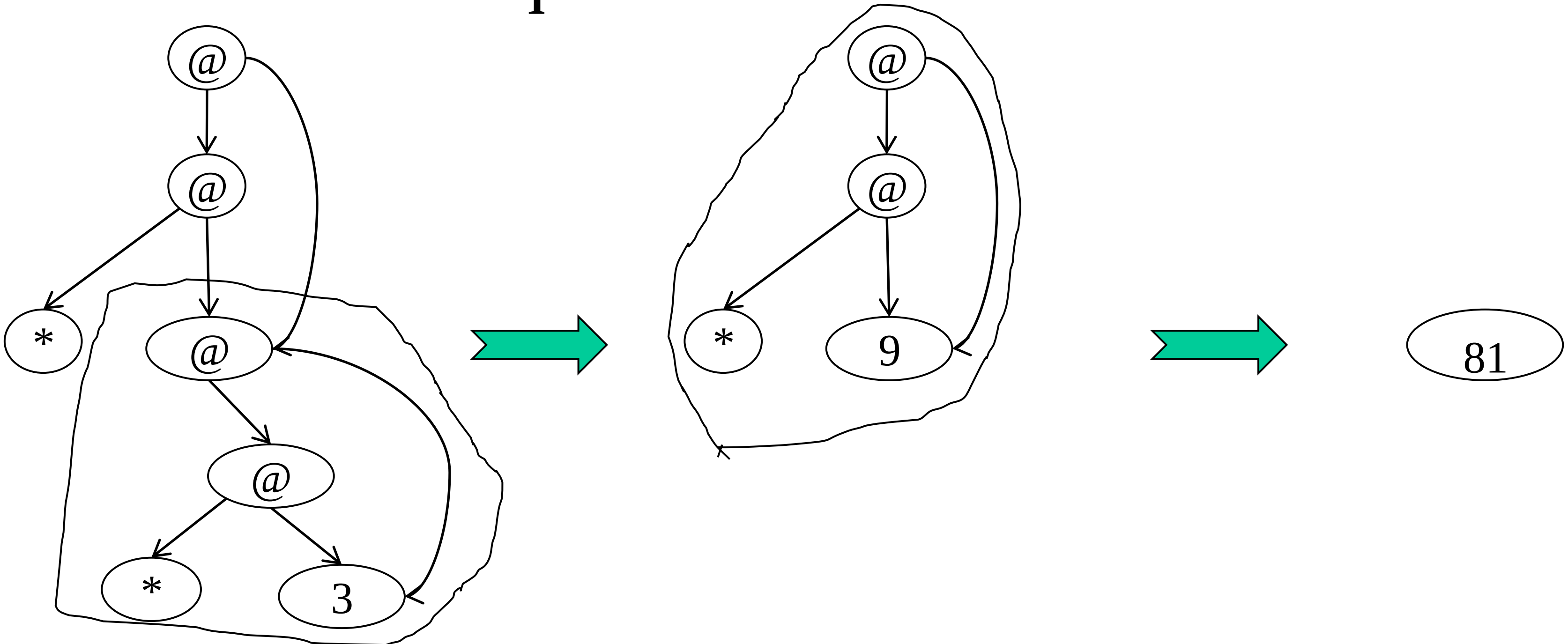


Another Example (cont3)

- let twice f x = f (f x)
  square n = n * n

  in

  twice square 3



Reduction Order

- At every point in execution redexes can be selected
- Start at the root
- If the root is not application node print the result
- If the root is an application node → its value is needed
  - Traverse the application spine to the left to find the function, say f
  - Check if the application spine contains all the arguments for f
    - No a Curried function is detected
    - Yes search and apply the redex

The reduction Engine

- Usually implemented in C
- Apply redexes
- Use a stack to match arguments
- Use Eval to apply redexes using function pointers
- Built in functions are part of the runtime system
- User defined functions are mapped into C in a straightforward way using Eval function
- Significant runtime overhead

C Header file

```
typedef enum {FUNC, NUM, NIL, CONS, APPL} node_type
typedef  struct  node *Pnode
typedef  Pnode (*unary) (Pnode *arg)
struct function_descriptor {
   int arity;
   const char * name;
   unary code ;
};
struct node_type {
   node_type tag;
   union {
     struct function_descriptor func;
     int num;
     struct {Pnode hd, Pnode tl ;} cons
     struct {Pnode fun; Pnode arg;} appl;
     } nd;
}
extern Pnode Func(int arity, const char *name, unary code);
extern Pnode Nil(int num);
…
```

Reducing the cost of graph reduction

- Shortcut reductions
- Detect lazy expressions which are always executed
  - Strict

Optimizing the functional core

- Boxing analysis
- Tail call elimination
- Accumulator translation
- Strictness analysis

Strictness Analysis

- A function is strict in an argument a if it needs the value of a in all executions
- Example safe_div a b = if (b == 0) then 0
  else a / b
  - strict in b
- Can be computed using dataflow equations

Limitations

- Usually the generated C code can be reasonably efficient
- Current strictness analysis works poorly for user-defined data structures and higher order functions

Summary

- Functional programs provide concise coding
- Compiled code compares with C code
- Successfully used in some commercial applications
  - F#, ERLANG
- Ideas used in imperative programs
- Less popular than imperative programs