Compiler construction lecture 13

Functional programming

Chapter 7

Functional programming

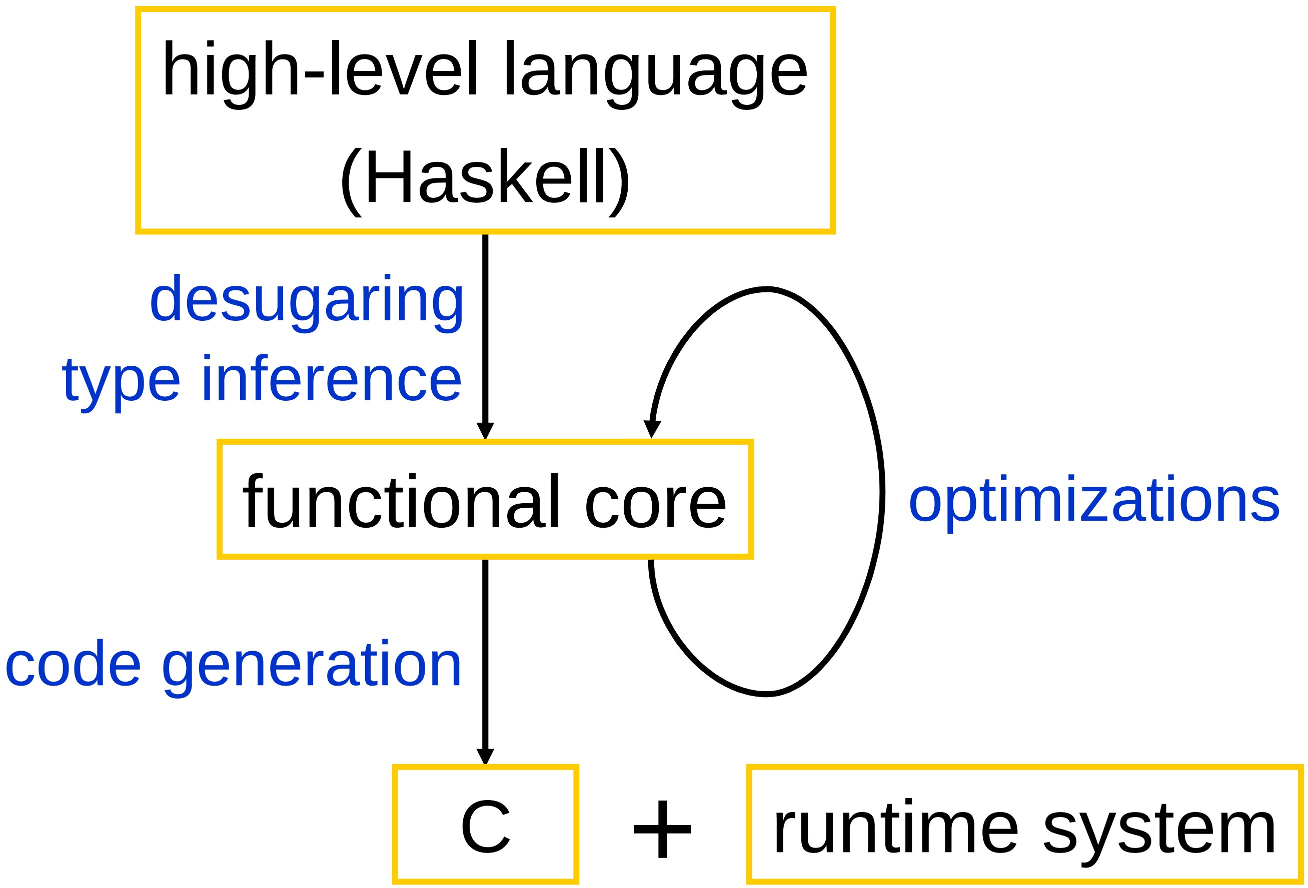
- languages:

  LISP, Scheme, ML, Miranda, Haskell

- features:
  - high abstraction level: what vs how, where, when
  - equational reasoning
  - functions as first class citizens

compiler must work harder!!

Overview of a typical functional compiler



Function application

- concise notation

Haskell	C
`f 11 13`	`f(11, 13)`

- precedence over all other operators

`f n+1`	`f(n) + 1`

Syntactic sugar

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

- offside rule: end-of-equation marking
- list notation: `[]` `[1,2,3]` `(1:(2:(3:[])))`
- pattern matching: case analysis of arguments
- list comprehension: mathematical sets

Polymorphic typing

- an expression is polymorphic if it

  ‘has many types’

- examples
  - empty list: []
  - list handling functions

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

Polymorphic type inference

```
map f []     = []
map f (x:xs) = f x : map f xs
```

$\Rightarrow$ `map :: ` t_1 `->` t_2 `->` t_3

```
map f []     = []
```

$\Rightarrow$ `map :: ` t_1 `-> [a] -> [b]`

```
map f (x:xs) = f x : map f xs
```

```
  x :: a
  f :: a -> b
map :: (a -> b) -> [a] -> [b]
```

Exercise (5 min.)

- infer the polymorphic type of the following higher-order function:

```
filter f []     = []
filter f (x:xs) = if not(f x) then  filter f xs
                  else x : (filter f xs)
```

Answers

Referential transparency

`f x` always denotes the same value

- advantage: high-level optimization is easy

```
g (f x) (f x)
```

```
let a = f x
in g a a
```

- disadvantage: no efficient in-place update

```
add_one []     = []
add_one (x:xs) = x+1 : add_one xs
```

Higher-order functions

- functions are first-class citizens
- higher-order functions accept functions as parameters and/or return a function as result
- functions may be created "on the fly"

$$D\,f = f' \text{ where } f'(x) = \lim_{h\downarrow 0} \frac{f(x+h) - f(x)}{h}$$

```
diff f = f_
         where
            f_ x = (f (x+h) - f x)/h
            h    = 0.0001
```

Currying: specialize functions

```
diff f = f_
where
  f_ x = (f (x+h) – f x)/h
  h    = 0.0001
```

```
deriv f x = (f (x+h) – f x)/h
            where
              h = 0.0001
```

Q: **`diff`** (unary function) $\equiv$ **`deriv`** (binary function)?

A: yes! $\boxed{\forall \mathtt{f},\ \forall \mathtt{x}\quad \mathtt{(diff\ f)\ x = deriv\ f\ x}}$

binary function $\equiv$ a unary function returning a unary function

$$f\ e_1 \ldots e_n \equiv (^n\ f\ e_1) \ldots e_n)$$

`(deriv square)` is a curried function

Lazy evaluation

An expression will only be evaluated when its value is needed to progress the computation

- additional expressive power (infinite lists)

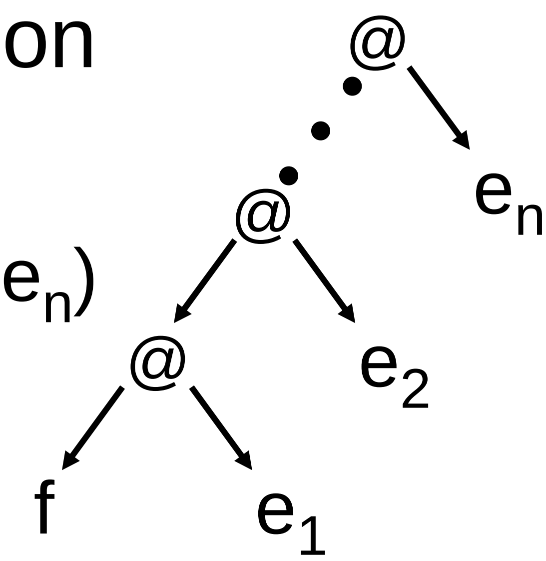
```
deriv f x = lim [(f (x+h) – f x)/h | h <- downto 0]
where
    downto x = [x + 1/2^n | n <- [1..]]
    lim (a:b:lst) = if abs (a/b – 1) < eps then b
                    else lim (b:lst)
```

- overhead for delaying/resuming computations

Graph reduction

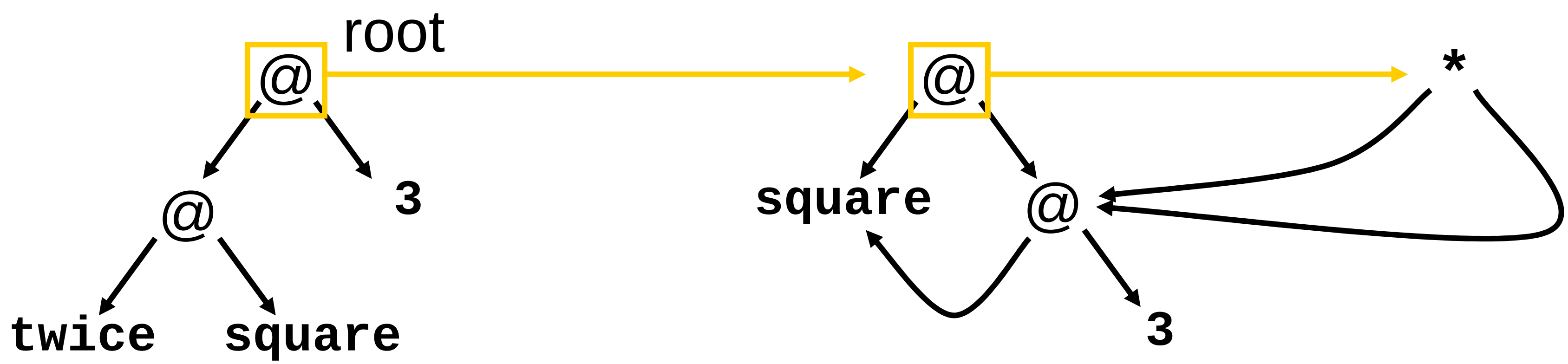
- implement h.o.f + lazy evaluation
- key: function application

  $f\ e_1 \ldots e_n \equiv ({}^n\ f\ @\ e_1)\ @\ e_2) \ldots @\ e_n)$

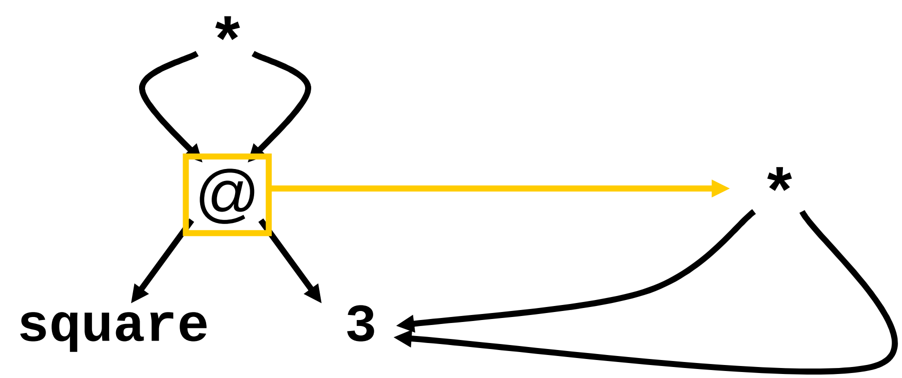


- execution (interpretation)
  - build graph for main expression
  - find reducible expression (redex ≡ func + args)
  - instantiate body (build graph for rhs)

Example

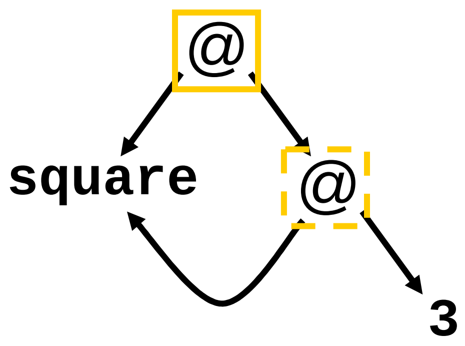


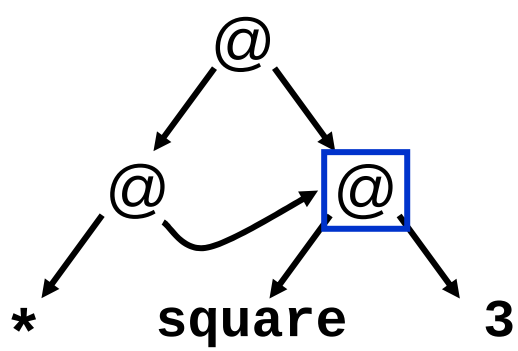

```
let
  twice f x = f (f x)
  square n = n*n
in
  twice square 3
```



Reduction order

- a graph may contain multiple redexes
- lazy evaluation: choose top-most @-node





- built-in operators (+,-,*, etc) may have strict arguments that must be evaluated => recursive invocation

Implementation

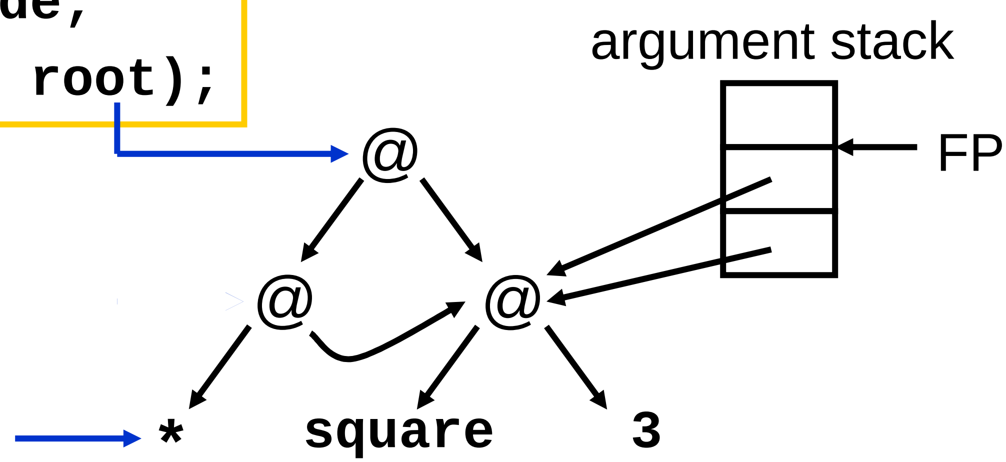
```
Pnode mul(Pnode arg[]) {
  Pnode a = eval(arg[0]);
  Pnode b = eval(arg[1]);

  return Num(a->nd.num * b->nd.num);
}
```

Graph reduction

- find next redex
- instantiate rhs
- update root

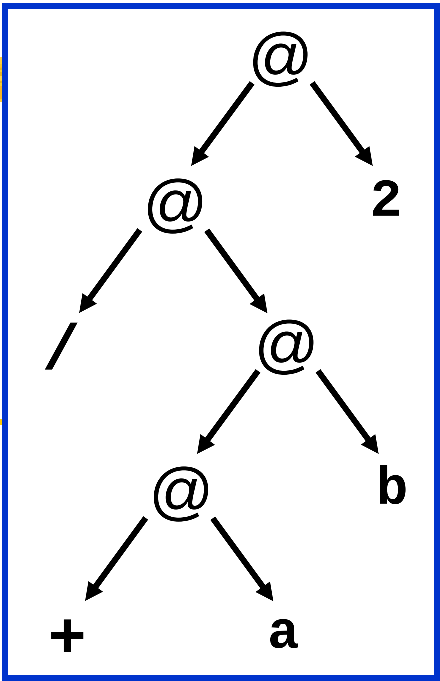
```
typedef struct node *Pnode;
extern Pnode eval( Pnode root);
```

- unwind application spine
  (f a_1 …a_n)
- call f, pass arguments in
  array (stack)



Code generation

```
average a b = (a+b) / 2
```

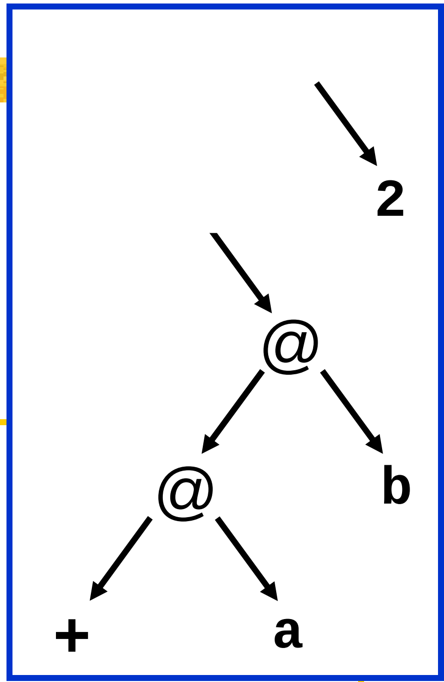



```
Pnode average(Pnode arg[]) {
  Pnode a = arg[0];
  Pnode b = arg[1];

  return Appl(Appl(fun_div,
                   Appl(Appl(fun_add,a),b)),
              Num(2));
}
```

Short-circuiting application spines

```
average a b = (a+b) / 2
```

```
Pnode average(Pnode arg[]) {
  Pnode a = arg[0];
  Pnode b = arg[1];

  return div(Appl(Appl(fun_add,a),b),
             Num(2));
}
```

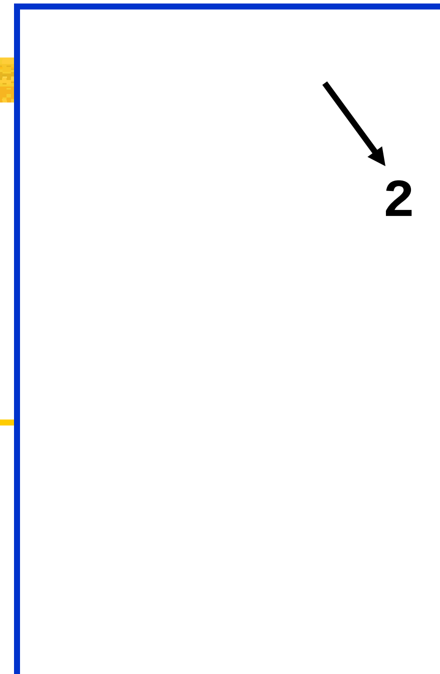


- call leftmost outermost function directly

Strict arguments

```
average a b = (a+b) / 2
```

```
Pnode average(Pnode arg[]) {
  Pnode a = arg[0];
  Pnode b = arg[1];

  return div(add(a,b), Num(2));
}
```



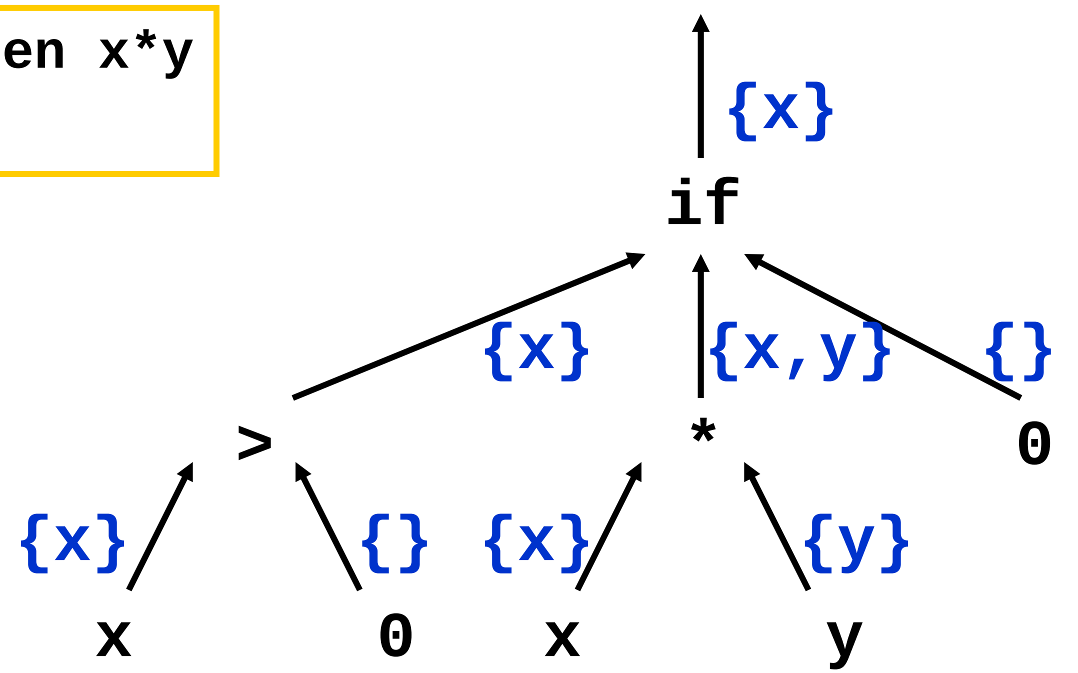
- evaluate expressions supplied to strict built-in functions immediately

Strictness analysis

user-defined functions:

- propagate sets of strict arguments up the AST

```
foo x y = if x>0 then x*y
          else 0
```

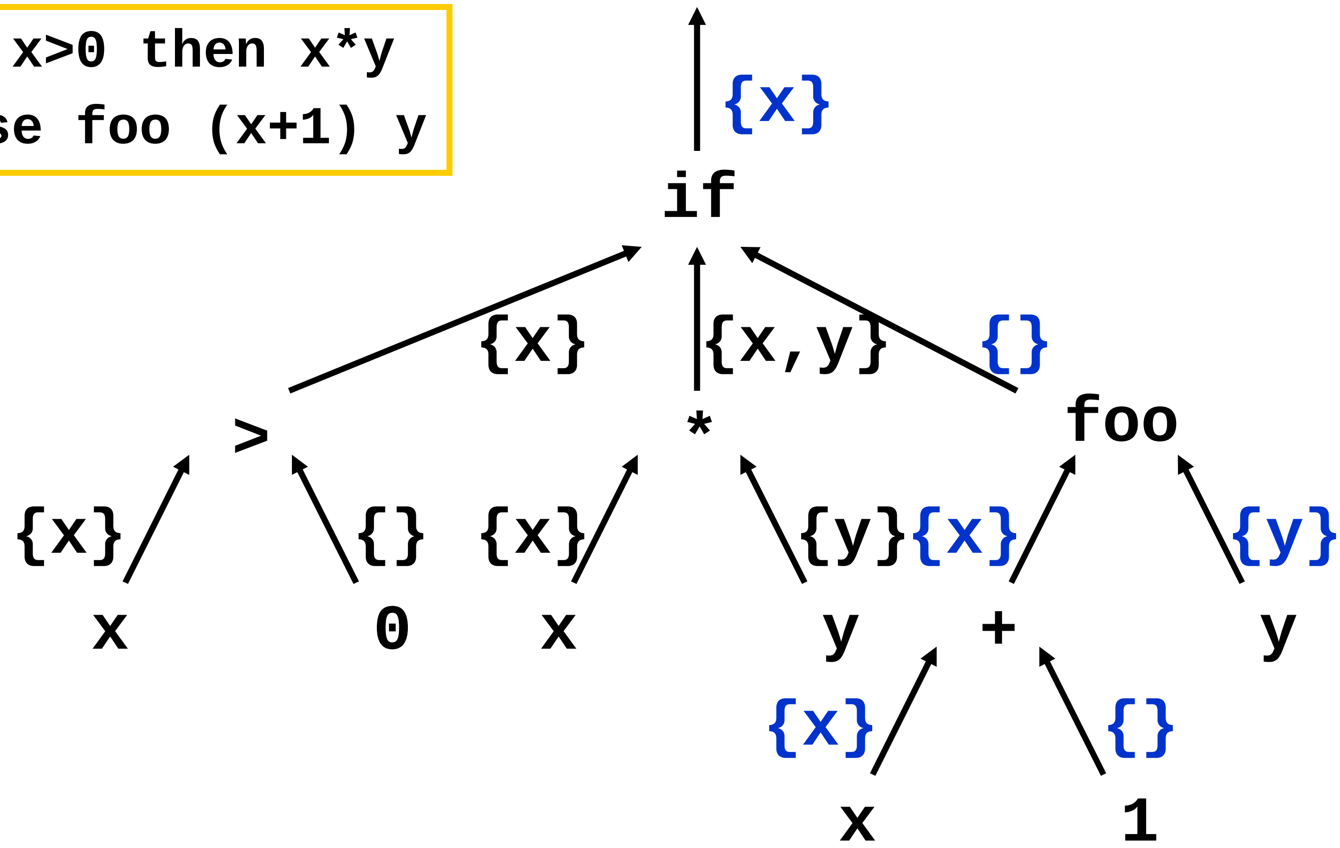


Strictness propagation

language construct	**propagated set**
$L \oplus R$	$L \cup R$
if C then T else E	$C \cup (T \cap E)$
fun_m @ A_1 @ … @ A_n	$\bigcup_{i=1}^{m}{}_{\text{strict(fun,i)}} A_i$, if $n \geq m$
…	…

Recursive functions

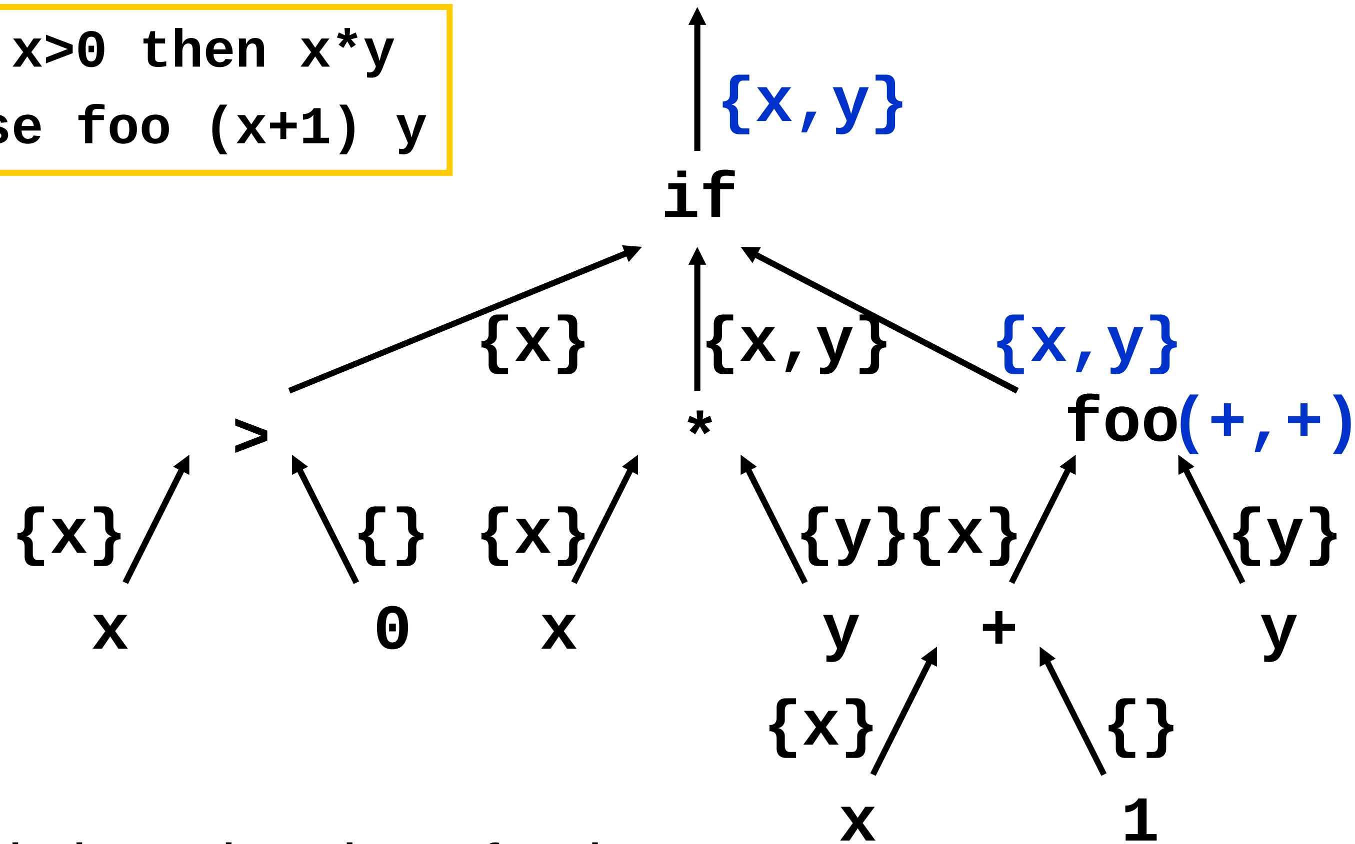
```
foo x y = if x>0 then x*y
          else foo (x+1) y
```



problem: conservative estimation of strictness

Recursive functions

```
foo x y = if x>0 then x*y
             else foo (x+1) y
```



solution: optimistic estimation of strictness

iterate until result equals assumption

Exercise (6 min.)

- infer the strict arguments of the following recursive function:

```
g x y 0 = x
g x y z = g y x (z-1)
```

- how many iterations are needed?

Answers

step	assumption	result
1		
2		
3		
4		

Summary

Haskell feature	**compiler phase**
offside rule	lexical analyzer
list notation list comprehension pattern matching	parser
polymorphic typing	semantic analyzer
referential transparency higher-order functions lazy evaluation	run-time system (graph reducer)