Overview

Programming Paradigms II

Overview

- Higher-order functions
  - What are they? Why they are useful?
  - Observations: Functions are data and can be treated like data.
  - Examples: Many of them are programming patterns.
- Some basic access-control concepts are also included

Introduction

A function is called **higher-order** if it takes a function as an argument or returns a function as a result.

twice f

Also a

function

```
twice :: (a → a) → a → a
twice f x = f (f x)
```

twice is higher-order because it takes a function as its first argument.

Why Are They Useful?

- **Common programming idioms** can be encoded as functions within the language itself.
- **Domain-specific languages** can be defined as collections of higher-order functions.
- **Algebraic properties** of higher-order functions can be used to reason about programs.

"foldr"

The End

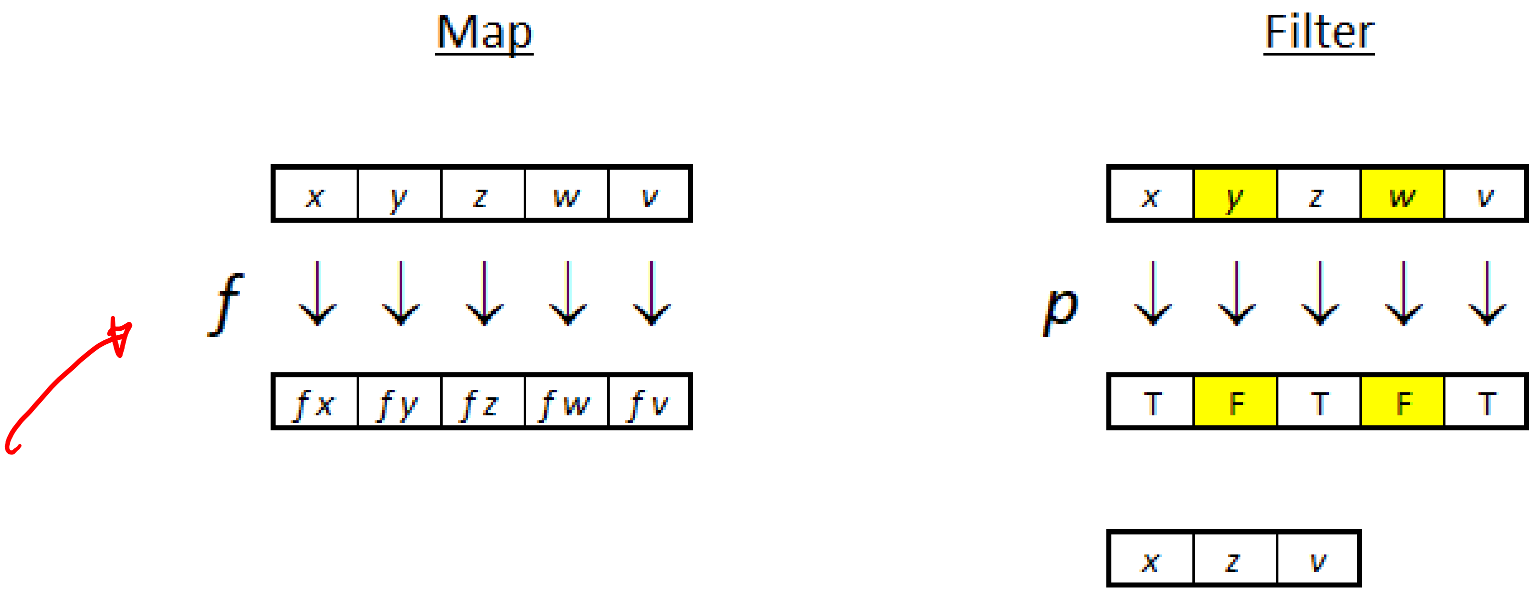
List Processing I

Maps and Filters

Maps and Filters

- map f $[x,y,z,w,v] = [f\ x\ ,\ f\ y\ ,\ f\ z\ ,\ f\ w\ ,\ f\ v]$
- filter p $[x,y,z,w,v] = [x\ \cancel{y}\ ,\ z\ ,\cancel{w}\ v]$

  $(p\ x = p\ z = p\ v =$ True; $p\ y = p\ w =$ False$)$



filter
those
elements
y, w

The Map Function

- The higher-order library function called **map** applies a function to every element of a list

```
map :: (a → b) → [a] → [b]
```

- For example:

```
> map (+1) [1,3,5,7]

[2,4,6,8]
```

- The map function can be defined in a particularly simple manner by using a list comprehension:

```
map f xs = [f x | x ← xs]
```

- Alternatively, for the purposes of proofs, the map function can also be defined by using recursion:

```
map f []     = []

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

The Filter Function

- The higher-order library function **filter** selects every element from a list that satisfies a predicate

```
filter :: (a → Bool) → [a] → [a]
```

predicate

- For example:

```
> filter even [1..10]

[2,4,6,8,10]
```

- Filter can be defined using a list comprehension:

```
filter p xs = [x | x ← xs, p x]
```

- Alternatively, it can be defined using recursion:

```
filter p [] = []
filter p (x:xs)
   | p x       = x : filter p xs
   | otherwise = filter p xs
```

The End

List Processing II

Using Lambda Functions

Lambda Functions

Example: lambda (λ) expressions (Hutton, Section 4.5)

- λ x y $\rightarrow$ x + y
- $\lambda\, f\, x \rightarrow f\, x$
- $\lambda\, f\, g \rightarrow (\lambda\, x \rightarrow f\, (g\, x))$

Why Are Lambdas Useful?

- Lambda expressions can be used to give a formal meaning to functions defined using **currying**
- For example:

add x y = x + y

means

add = $\lambda x \rightarrow (\lambda y \rightarrow x + y)$

- Lambda expressions are also useful when defining functions that return **functions as results**
- For example:

```
const :: a → b → a
const x _ = x
```

is more naturally defined by

```
const :: a → (b → a)
const x = λ_ → x
```

- Lambda expressions can be used to avoid naming functions that are only **referenced once**
- For example:

```
odds n = map f [0..n-1]
         where
            f x = x*2 + 1
```

can be simplified to

```
odds n = map (λx → x*2 + 1) [0..n-1]
```

Using Lambda Functions

- Using lambda functions with higher-order functions
- Example: *flip* takes a function *f* as input
  (e.g., $f\ x\ y = x - y$), returns a *function flip f* such that

$$(flip\ f)\ x\ y = f\ y\ x$$

- As a lambda function: $flip\ f = (\lambda\ x\ y \rightarrow f\ y\ x)$

List Processing II

The End

Programming Patterns I

The Composition Operator

The Composition Operator

- The library function (.) returns the **composition** of two functions as a single function

```
(.) :: (b → c) → (a → b) → (a → c)
f . g = λx → f (g x)
```

- For example:

```
odd :: Int → Bool
odd = not . even
```

Examples

- *twice f = f . f*
- *sumsqreven = sum . map (^2) . filter even*

[1, 2, 3, 4, 5]

filter even

[1, 3, 5]

map(^2)

[1, 9, 25]

1+9+25

= 35

Programming Patterns I: The Composition Operator

The End

Programming Patterns I

Other Common Patterns

- The library function **all** decides if every element of a list satisfies a given predicate

```
all :: (a → Bool) → [a] → Bool
all p xs = and [p x | x ← xs]
```

- For example:

```
> all even [2,4,6,8,10]

True
```

- Dually, the library function **any** decides if at least one element of a list satisfies a predicate

```
any :: (a → Bool) → [a] → Bool
any p xs = or [p x | x ← xs]
```

- For example:

```
> any (== ' ') "abc def"

True
```

- The library function **takeWhile** selects elements from a list while a predicate holds of all the elements

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p [] = []
takeWhile p (x:xs)
   | p x       = x : takeWhile p xs
   | otherwise = []
```

- For example:

```
> takeWhile (/= ' ') "abc def"

"abc"
```

- Dually, the function **dropWhile** removes elements while a predicate holds all the elements

```
dropWhile :: (a → Bool) → [a] → [a]
dropWhile p [] = []
dropWhile p (x:xs)
   | p x       = dropWhile p xs
   | otherwise = x:xs
```

- For example:

```
> dropWhile (== ' ' ) "   abc"

"abc"
```

The End

Programming Patterns II

Recursion Operators

The Foldr Function

A number of functions on lists can be defined using the following simple pattern of recursion:

$$
\begin{aligned}
f\ [\,] &= v \\
f\ (x{:}xs) &= x \oplus f\ xs
\end{aligned}
$$

f maps the empty list to some value v, and any nonempty list to some function ⊕ applied to its head and f of its tail.

For example:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

$v = 0$

$\oplus = +$

```
product []     = 1
product (x:xs) = x * product xs
```

$v = 1$

$\oplus = *$

```
and []     = True
and (x:xs) = x && and xs
```

$v = \text{True}$

$\oplus = \&\&$

- The higher-order library function **foldr** (fold right) encapsulates this simple pattern of recursion, with the function ⊕ and the value v as arguments
- For example:

"Sum function"

```
sum = foldr (+) 0

product = foldr (*) 1

or = foldr (||) False

and = foldr (&&) True
```

sum [1,2,3] =

1+2+3 = 6

- Foldr itself can be defined using recursion:

```
foldr :: (a → b → b) → b → [a] → b
foldr f v []       = v
foldr f v (x:xs) = f x (foldr f v xs)
```

- However, it is best to think of foldr **nonrecursively,** as simultaneously replacing each (:) in a list by a given function and [] by a given value

For example:

sum [1,2,3]

=

foldr (+) 0 [1,2,3]

=

foldr (+) 0 (1:(2:(3:[])))

=

1+(2+(3+0))

=

6

Replace each (:) by (+) and [] by 0.

For example:

```
  product [1,2,3]
=
  foldr (*) 1 [1,2,3]
=
  foldr (*) 1 (1:(2:(3:[])))
=
  1*(2*(3*1))
=
  6
```

Replace each (:) by (*) and [] by 1.

Other foldr Examples

- Even though foldr encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected
- Recall the length function:

```
length :: [a] → Int
length []     = 0
length (_:xs) = 1 + length xs
```

- For example:

```
  length [1,2,3]
=
  length (1:(2:(3:[])))
=
  1+(1+(1+0))
=
  3
```

Replace each (:) by λ_ n → 1+n and [] by 0.

- Hence, we have:

λ _ n → 1 + n

```
length = foldr (λ_ n → 1+n) 0
```

initial/starting value]

- Now recall the reverse function:

*

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

- For example:

```
  reverse [1,2,3]
=
  reverse (1:(2:(3:[])))
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

Replace each (:) by λx xs → xs ++ [x] and [] by [].

← unpack the list [1,2,3]

← "Guess" what's definition of the operator

← "Trial & Error"

- Hence, we have:

```
reverse = foldr (λx xs → xs ++ [x]) []
```

- Finally, we note that the append function (++) has a particularly compact definition using foldr:

```
(++ ys) = foldr (:) ys
```

Replace each (:) by (:) and [] by ys.

Practice

Redefine map f and filter p using foldr

map f = foldr ((:).f) ([])

operator   initial value

filter p = foldr ( ) ([])

[[\x xs → if p x then x:xs else xs]]

A Variant of Foldr: Foldl

- The function foldl (“*fold from the left* ”)

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v []        = v
foldl f v (x:xs) = foldl f (f v x) xs
```

- How foldl works (algebraically)

```
foldl (#) v [x0,x1,...,xn] = (... ((v # x0) # x1) ...) # xn
```

$\text{foldl}\ (\#)\ v\ [x_0, x_1, x_2] = (((v \# x_0) \# x_1) \# x_2$

$\text{foldr}\ (\#)\ v\ [x_0, x_1, x_2] = x_0 \# (x_1 \# (x_2 \# v))$

Foldl: "Fold from the Left"

- length :: [a] → Int
- length :: foldl (   ) (   )
- Example

= ← operator

↲ ← initial value ↲

foldl (\ n _ → n+1) (0)

← defines the operator

↲ ← initial value ↲

Try length [1, 2] & see if you agree.

Practice: Define Reverse

```
reverse :: [a] -> [a]
reverse = foldl (\xs x -> x:xs) []
```

Try reverse [1,2] following definition above.

Programming Patterns II: Recursion Operators

The End

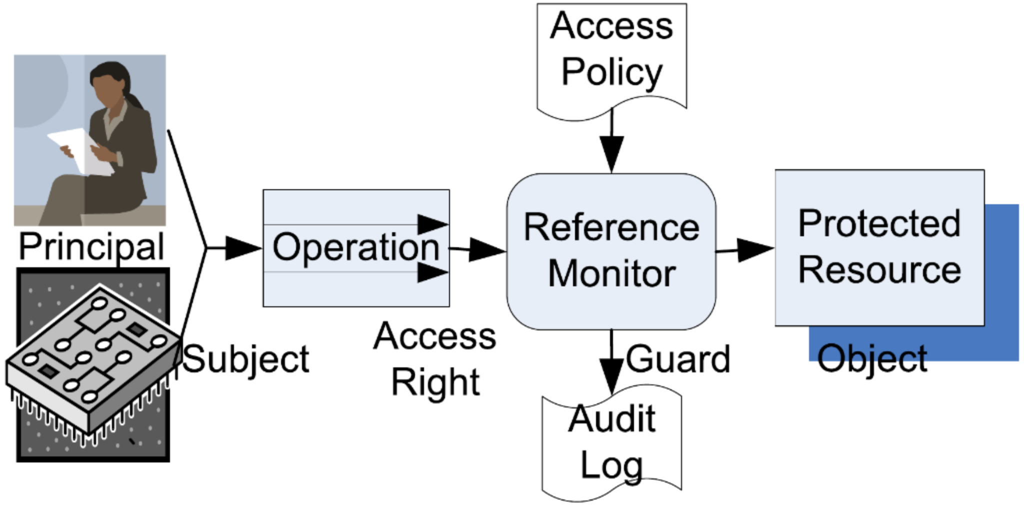
Basic Access-Control Concepts

Reference Monitors, Tickets

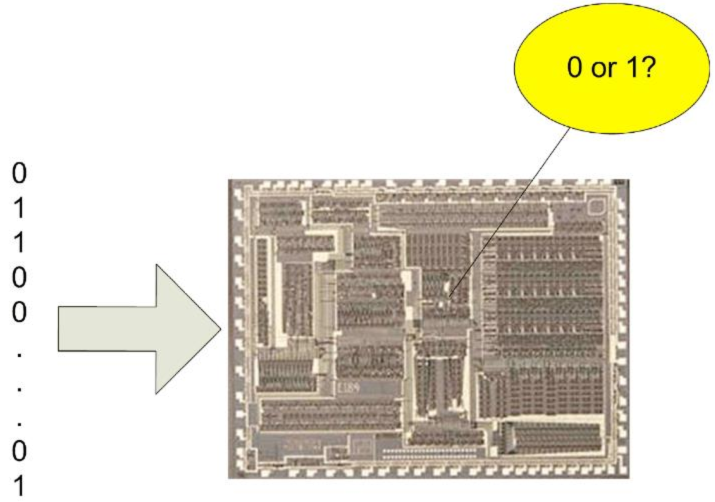
Access Control

Access control is central to security:

Who should be granted what access to which objects under what circumstances?

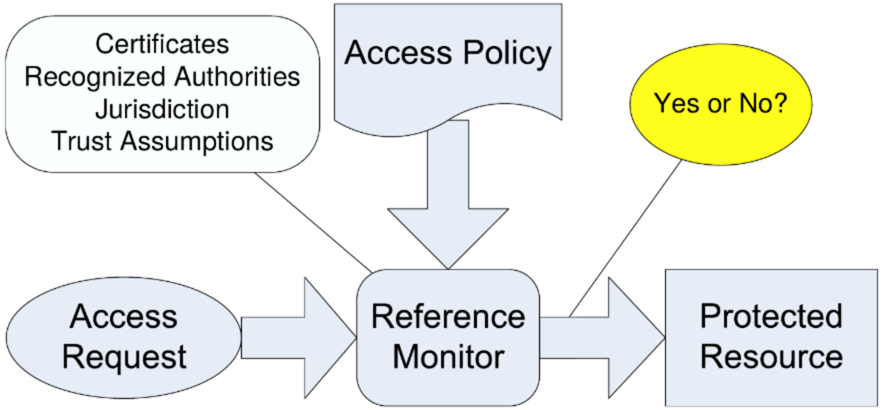


Expectations of Hardware Engineers



- *Mathematical derivation* of behavior
- Inability to do so is interpreted as **incompetence**

Our Position on Security and Trust



- *Mathematical derivation* of behavior given
  - Requests, policies, certifications, jurisdiction of authorities, and trust assumptions
- Inability to do so is interpreted as **incompetence**

Rigorous policy-based design and verification

Should I say "yes"? It depends on:

The access request: What is requested and by whom?

$$\text{Commander says } \langle \mathit{launch}, \mathit{missile} \rangle$$

The access policy: Who is authorized?

$$(\text{Commander} \ \& \ \text{Sub-commander}) \text{ controls } \langle \mathit{launch}, \mathit{missile} \rangle$$

Trust assumptions: Who is trusted or what is taken for granted? What are symbols or tokens of authority? Jurisdiction?

$$\text{National Command Authority controls}$$
$$((\text{Commander} \ \& \ \text{Sub-commander}) \text{ controls } \langle \mathit{launch}, \mathit{missile} \rangle)$$

$$\mathit{Key}_{\mathit{NCA}} \Rightarrow \text{National Command Authority}$$

Central Concepts

We focus on four central concepts:

1. **Reference monitors:** the guards protecting objects
2. **Tickets:** an unforgeable indicator of a principal's *capability* or right to access an object
3. **Access control lists:** a list of principals (with their rights) to access an object
4. **Authentication:** the process of identifying a principal (i.e., the source of a request)

Reference monitors enforce access control policies using concepts of tickets, access-control lists, or some combination of both.

Tickets: Unforgeable access tokens

- Simple ticket:

$$\text{Ticket says } (\mathit{Subject} \text{ controls } \langle \mathit{Access\ Right},\ \mathit{Object} \rangle)$$

- Simple access-control policy (jurisdiction & policy statement):

$$\text{Authority controls } (\mathit{Subject} \text{ controls } \langle \mathit{AccessRight}, \mathit{Object} \rangle)$$
$$\text{Authority says } (\mathit{Subject} \text{ controls } \langle \mathit{AccessRight}, \mathit{Object} \rangle)$$

- Trust assumption:

$$\text{Ticket} \Rightarrow \text{Authority}$$

- General form of a subject making a request using a ticket:

$$\mathit{Subject} \text{ says } \langle \mathit{AccessRight}, \mathit{Object} \rangle$$

Derived Inference Rule

We can capture ticket use as a derived rule:

$$\textit{Ticket Rule}\ \frac{\textit{subject}\ \textsf{says}\ \varphi \qquad \textit{authority}\ \textsf{controls}\ (\textit{subject}\ \textsf{controls}\ \varphi) \qquad \textit{ticket} \Rightarrow \textit{authority} \qquad \textit{ticket}\ \textsf{says}\ (\textit{subject}\ \textsf{controls}\ \varphi)}{\varphi}$$

- Advantages
  - Simplifies and reduces size of proofs
  - Soundness
  - Specifies what specific reference monitor needs to do (i.e., a checklist)

Exercise: Derive the Ticket Rule

Your proof should begin with:

1.	$\mathit{subject}$ says φ	Access request
2.	$\mathit{authority}$ controls $(\mathit{subject}$ controls $\varphi)$	Access policy
3.	$\mathit{ticket} \Rightarrow \mathit{authority}$	Trust assumption
4.	ticket says $(\mathit{subject}$ controls $\varphi)$	Ticket

Goal: derive φ

Ticket Rule

Proof tree construction

② Control.

① Subject says φ $\boxed{\text{Subjects controls}}\ \boxed{\phi}$ Controls

φ conclusion

A Derivation

1.	*subject* says φ	Access request
2.	*authority* controls (*subject* controls φ)	Access policy
3.	*ticket* $\Rightarrow$ *authority*	Trust assumption
4.	*ticket* says (*subject* controls φ)	Ticket
5.	*authority* says (*subject* controls φ)	3, 4 speaks for
6.	*subject* controls φ	2, 5 controls
7.	φ	6, 1 controls

$$\frac{\boxed{3} \quad \boxed{4}}{5}\ \text{speaks for}$$

$$\frac{\boxed{2} \quad 5}{6}\ \text{controls.}$$

$$\frac{\boxed{1} \quad 6}{7}\ \text{controls}$$

Basic Access-Control Concepts: Reference Monitors, Tickets

The End

Basic Access-Control Concepts

Access-Control Mechanisms: Tickets

A Derivation

1. *subject* says φ	Access request
2. *authority* controls (*subject* controls φ)	Access policy
3. *ticket* $\Rightarrow$ *authority*	Trust assumption
4. *ticket* says (*subject* controls φ)	Ticket
5. *authority* says (*subject* controls φ)	3, 4 speaks for
6. *subject* controls φ	2, 5 controls
7. φ	6, 1 controls

Ticket Rule

- The inference rule

$$\textit{Ticket Rule} \quad \frac{\begin{array}{cc} \textit{subject} \text{ says } \varphi & \textit{authority} \text{ controls } (\textit{subject} \text{ controls } \varphi) \\ \textit{ticket} \Rightarrow \textit{authority} & \textit{ticket} \text{ says } (\textit{subject} \text{ controls } \varphi) \end{array}}{\varphi}$$

- Hypotheses used are:

$\textit{subject} \text{ says } \varphi$	Access request
$\textit{authority} \text{ controls } (\textit{subject} \text{ controls } \varphi)$	Access policy
$\textit{ticket} \Rightarrow \textit{authority}$	Trust assumption
$\textit{ticket} \text{ says } (\textit{subject} \text{ controls } \varphi)$	Ticket

Example Using Tickets

Tina has an airplane ticket that assigns her to seat 25D on SmoothAir Flight #1.
When Tina's row is called, she presents her ticket to the gate agent for flight #1. What is the justification for granting her access to board?

1.	Tina says $\langle$seat 25D, flight #1$\rangle$	Tina's request
2.	SmoothAir controls (Tina controls $\langle$seat 25D, flight #1$\rangle$)	Access policy
3.	ticket $\Rightarrow$ SmoothAir	Trust assumption
4.	ticket says (Tina controls $\langle$seat 25D, flight #1$\rangle$)	Tina's ticket
5.	$\langle$seat 25D, flight #1$\rangle$	1, 2, 3, 4 ticket rule

Exercise

Exercise 4.2.1 *Suppose a theater ticket is sold by the box office to a patron to see* Gone with the Wind *in Theater 5 at 7:30 p.m. Using the access-control logic, describe the patron's request, the access-control policy of the theater, the trust assumptions, and movie ticket. Based on your descriptions, formally justify admitting the patron to see the movie.*

Exercise 4.2.1

Discussions

Trust Assumption:

ticket $\Rightarrow$ Box Office

speaks for

Patron Request:

Patron says (view GWTW in Thr 5 at 7:30 pm)

Exercise 4.2.1 (cont.)

Discussions (cont.)

The policy :-

Box office controls

(Patron controls (view GWTH in Thr 5
at 7:30 pm)

Ticket :-

ticket says (Patron controls)

Basic Access-Control Concepts: Access-Control Mechanisms: Tickets

The End

Weekly Summary

Programming Paradigms II

Overview

- Higher-order functions
  - What are they? Why they are useful?
  - Observations: Functions are data and can be treated like data.
  - Examples: Many of them are programming patterns.
- Some basic access-control concepts are also included

Higher-Order Functions

- Functions that take other function(s) as input
- Utilize lambda expressions to represent input functions
- Capture many programming patterns, including basic list processing, function composition, iteration, and recursion
- Interesting properties (e.g., algebraic properties of foldr) can be used to reason about programs

Basic Access-Control Concepts

- Introduce and discuss reference monitors, the guards for computer and information systems.
- Reference monitors provide the context for our studies in security mechanism. The fundamental concepts behind can be expressed in access-control logic. Derived inference rules can be obtained, and it allows proper reasoning about the systems.

Weekly Summary

The End