Programming Paradigms

Lecture 1. Programming paradigms overview. Lambda calculus

Outline

- What is a programming paradigm?
- Declarative vs Imperative
- Overview of common programming paradigms
- Functional Programming
- Lambda calculus recap

What is a programming paradigm?

A **programming paradigm** is a style of programming.

Programming languages can make it easier to write programs in some programming paradigms, but not others.

Declarative vs Imperative

Imagine a program that prints out a list of students, who get an A in a course.

Declarative vs Imperative

Imagine a program that prints out a list of students, who get an A in a course.

```
result = [];
for (i = 0; i < length(students); i++) {
    student = students[i];
    if (student.points >= 85) {
        addToList(result, toUpper(student.name));
    }
}
return sort(result);
```

Declarative vs Imperative

Imagine a program that prints out a list of students, who get an A in a course.

```
select toUpper(student.name) as name
from students
where students.points >= 85
order by name
```

Declarative vs Imperative

Imagine a program that prints out a list of students, who get an A in a course.

```
students_with_A(Result)
   :- setof(Name,
            (student(Name, Points), Points >= 85),
            Result).
```

Declarative vs Imperative

Imagine a program that prints out a list of students, who get an A in a course.

```
result = []
for i in range(len(students)):
  student = students[i]
  if student.points >= 85:
    result.append(toUpper(student.name))

return sorted(result)
```

Declarative vs Imperative

Imagine a program that prints out a list of students, who get an A in a course.

```
sorted([student.name for student in students if student.points >= 85])
```

Declarative vs Imperative

Imagine a program that prints out a list of students, who get an A in a course.

```
def students_with_A(students):
    if len(students) == 0: return []

    student = students[0]
    result = students_with_A(students[1:])
    if student.points >= 85:
        result.append(student.name)
    return result
```

Declarative vs Imperative

Imagine a program that prints out a list of students, who get an A in a course.

```
(sort
 (map name
      (filter (lambda (student) (>= (points student) 85))
              students))
 <=)
```

Declarative vs Imperative

Imagine a program that prints out a list of students, who get an A in a course.

```
do
  put []
  forM_ [0 .. length students - 1] $ \i -> do
    let student = students ! i
    when (points student >= 85) $ do
      modify $ \result ->
        Vector.cons (name student) result
  result <- get
  return (sort result)
```

Declarative vs Imperative

Imagine a program that prints out a list of students, who get an A in a course.

```
let step result i =
      let student = students ! i
       in if (points student >= 85)
             then Vector.cons (name student) result
             else result
 in foldl' step [] students
```

Declarative vs Imperative

Imagine a program that prints out a list of students, who get an A in a course.

```
students
   .filter(function(student) {return student.points >= 85;})
   .map(function(student) {return student.name; })
   .sort()
```

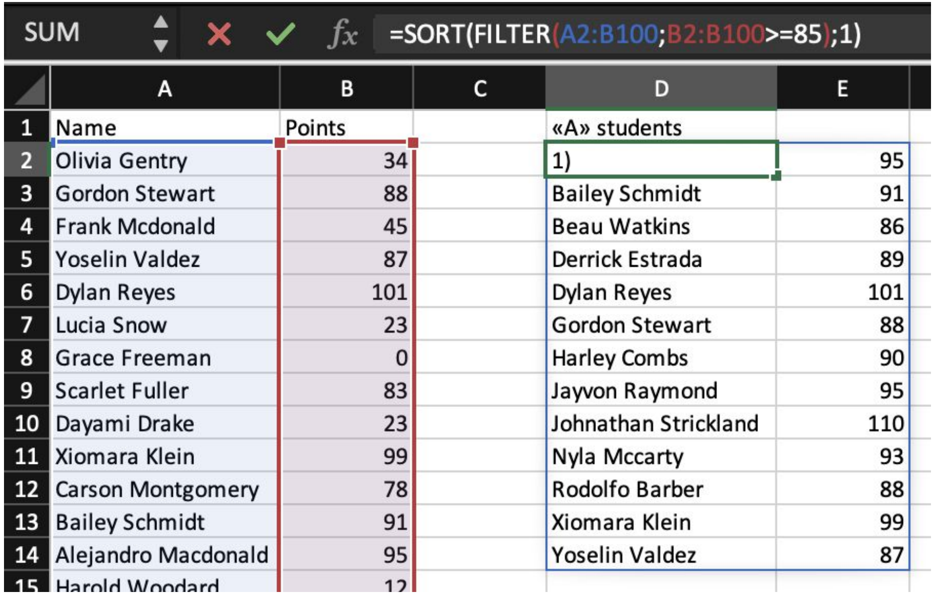
Declarative vs Imperative

Imagine a program that prints out a list of students, who get an A in a course.

```
[ name student | student <- students, points student >= 85 ]
```

Declarative vs Imperative

Imagine a program that prints out a list of students, who get an A in a course.


SUM =SORT(FILTER(A2:B100;B2:B100>=85);1)

	A	B	C	D	E
1	Name	Points		«A» students	
2	Olivia Gentry	34		1)	95
3	Gordon Stewart	88		Bailey Schmidt	91
4	Frank Mcdonald	45		Beau Watkins	86
5	Yoselin Valdez	87		Derrick Estrada	89
6	Dylan Reyes	101		Dylan Reyes	101
7	Lucia Snow	23		Gordon Stewart	88
8	Grace Freeman	0		Harley Combs	90
9	Scarlet Fuller	83		Jayvon Raymond	95
10	Dayami Drake	23		Johnathan Strickland	110
11	Xiomara Klein	99		Nyla Mccarty	93
12	Carson Montgomery	78		Rodolfo Barber	88
13	Bailey Schmidt	91		Xiomara Klein	99
14	Alejandro Macdonald	95		Yoselin Valdez	87
15	Harold Woodard	12			

`SORT(FILTER(A2:B100;B2:B100>=85);1)`

Common programming paradigms: an overview

1. Imperative
2. Structural programming
3. Procedural programming
4. Object-oriented programming
5. Declarative
6. Functional programming
7. Logical programming
8. Array programming
9. and more…

https://cs.lmu.edu/~ray/notes/paradigms/

Common programming paradigms: an overview

1. **Imperative**
2. Structural programming
3. Procedural programming
4. Object-oriented programming
5. **Declarative**
6. **Functional programming**
7. **Logical programming**
8. Array programming
9. and more…

https://cs.lmu.edu/~ray/notes/paradigms/

Functional Programming

Why Functional Programming Matters

John Hughes
The University, Glasgow

Abstract

As software becomes more and more complex, it is more and more important to structure it well. Well-structured software is easy to write and to debug, and provides a collection of modules that can be reused to reduce future programming costs. In this paper we show that two features of functional languages in particular, higher-order functions and lazy evaluation, can contribute significantly to modularity. As examples, we manipulate lists and trees, program several numerical algorithms, and implement the alpha-beta heuristic (an algorithm from Artificial Intelligence used in game-playing programs). We conclude that since modularity is the key to successful programming, functional programming offers important advantages for software development.

https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf

Functional features in non-functional languages

1. **map**, **filter**, **reduce** in many languages
2. **list comprehensions** in Python, LINQ in C#
3. **lambda expressions** in Java 8+, **anonymous functions**
4. **enumeration cases** in Swift, enums in Rust (already in Go?)
5. **generics** in Java, C#
6. etc.

Lambda calculus — Basics

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

Lambda calculus — Basics

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

factorial = λn. if n=0 then 1 else n * factorial(n-1)
```

Lambda calculus — Basics

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

factorial = λn. if n=0 then 1 else n * factorial(n-1)

factorial 0
```

Lambda calculus — Basics

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

factorial = λn. if n=0 then 1 else n * factorial(n-1)

factorial 0
= (λn. if n=0 then 1 else n * factorial(n-1)) 0
```

Lambda calculus — Basics

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

factorial = λn. if n=0 then 1 else n * factorial(n-1)

factorial 0
= (λn. if n=0 then 1 else n * factorial(n-1)) 0
= if 0=0 then 1 else 0 * factorial(0-1)
```

Lambda calculus — Basics

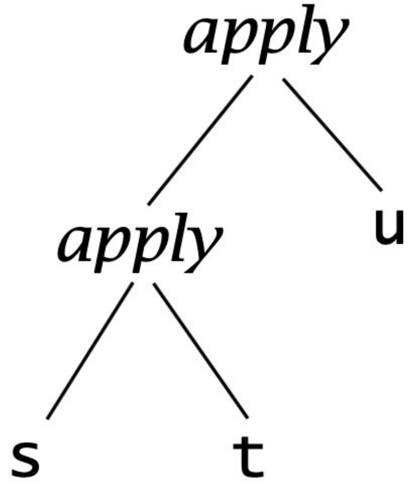
```
factorial(n) = if n=0 then 1 else n * factorial(n-1)

factorial = λn. if n=0 then 1 else n * factorial(n-1)

factorial 0
= (λn. if n=0 then 1 else n * factorial(n-1)) 0
= if 0=0 then 1 else 0 * factorial(0-1)
= 1
```

Lambda calculus — Syntax

t	::=		*terms:*
		x	*variable*
		λx.t	*abstraction*
		t t	*application*

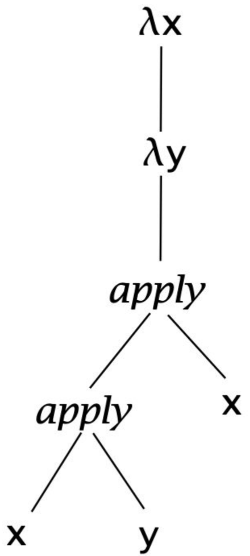
Lambda calculus — Syntax conventions

`s t u` is the same as `(s t) u`



Lambda calculus — Syntax conventions

`λx. λy. x y x` is the same as `λx. (λy. ((x y) x))`



Lambda calculus — Scopes

Occurrence of a variable `x` is **bound** when it occurs in a body `t` of `λx.t`

Occurrence of a variable `x` is **free** if it is not bound by any λ-abstraction

Lambda calculus — Scopes

Occurrence of a variable x is **bound** when it occurs in a body `t` of `λx.t`

Occurrence of a variable x is **free** if it is not bound by any λ-abstraction

```
λy. x y
```

Lambda calculus — Scopes

Occurrence of a variable x is **bound** when it occurs in a body `t` of `λx.t`

Occurrence of a variable x is **free** if it is not bound by any λ-abstraction

```
λy. x y
```

free

Lambda calculus — Scopes

Occurrence of a variable x is **bound** when it occurs in a body `t` of `λx.t`

Occurrence of a variable x is **free** if it is not bound by any λ-abstraction

$$\lambda y.\ x\ y$$

bound

Lambda calculus — Scopes

Occurrence of a variable x is **bound** when it occurs in a body `t` of `λx.t`

Occurrence of a variable x is **free** if it is not bound by any λ-abstraction

```
λz. λx. λy. x (y z)
```

Lambda calculus — Scopes

Occurrence of a variable x is **bound** when it occurs in a body `t` of `λx.t`

Occurrence of a variable x is **free** if it is not bound by any λ-abstraction

```
λz. λx. λy. x (y z)
```

bound

Lambda calculus — Scopes

Occurrence of a variable x is **bound** when it occurs in a body `t` of `λx.t`

Occurrence of a variable x is **free** if it is not bound by any λ-abstraction

$$\lambda z.\ \lambda x.\ \lambda y.\ x\ (y\ z)$$

bound

Lambda calculus — Scopes

Occurrence of a variable x is **bound** when it occurs in a body `t` of `λx.t`

Occurrence of a variable x is **free** if it is not bound by any λ-abstraction

```
λz. λx. λy. x (y z)
```

bound

Lambda calculus — Scopes

A term is **closed** if it has no free variables.

$$\mathtt{id} = \lambda x.\ x$$

Lambda calculus — Operational Semantics

$$(\lambda x.t)u \to [x \mapsto u]t$$

`x` is a metavariable ranging across all variables

`t` and `u` are metavariables ranging across all terms

Lambda calculus — Operational Semantics

$$(\lambda x.t)u \rightarrow \underbrace{[x \mapsto u]t}$$

Replace every free occurrence of x in term `t` with term u

Lambda calculus — Operational Semantics

$$(\lambda x.t)u \rightarrow [x \mapsto u]t$$

$$(\lambda x.x)y \rightarrow [x \mapsto y]x$$

Lambda calculus — Operational Semantics

$$(\lambda x.t)u \rightarrow [x \mapsto u]t$$

$$(\lambda x.x)y \rightarrow [x \mapsto y]x$$

$$= y$$

Lambda calculus — Operational Semantics

$$(\lambda x.t)u \rightarrow [x \mapsto u]t$$

$$(\lambda x.\ x\ (\lambda x.x))\ (u\ r)$$

Lambda calculus — Operational Semantics

$$(\lambda x.t)u \longrightarrow [x \mapsto u]t$$

$$(\lambda x.\ x\ (\lambda x.x))\ (u\ r)$$

Lambda calculus — Operational Semantics

$$(\lambda x.t)u \rightarrow [x \mapsto u]t$$

$$(\lambda x.\ x\ (\lambda x.x))\ (u\ r)$$
$$\rightarrow [x \mapsto (u\ r)]\ (x\ (\lambda x.x))$$

Lambda calculus — Operational Semantics

$$(\lambda x.t)u \to [x \mapsto u]t$$

$$(\lambda x.\ x\ (\lambda x.x))\ (u\ r)$$
$$\to [x \mapsto (u\ r)]\ (x\ (\lambda x.x))$$

Lambda calculus — Operational Semantics

$$(\lambda x.t)u \to [x \mapsto u]t$$

$$\begin{aligned}
&(\lambda x.\ x\ (\lambda x.x))\ (u\ r) \\
&\to [x \mapsto (u\ r)]\ (x\ (\lambda x.x)) \\
&= (u\ r)\ (\lambda x.x)
\end{aligned}$$

Lambda calculus — Reduction Strategies

Full beta-reduction — any redex can be reduced at any time.

```
id (id (λz. id z))
```

Lambda calculus — Reduction Strategies

Full beta-reduction — any redex can be reduced at any time.

```
id (id (λz. id z))
id (id (λz. id z))
id (id (λz. id z))
```

Lambda calculus — Reduction Strategies

Full beta-reduction — any redex can be reduced at any time.

```
id (id (λz. id z))
→ id (id (λz. z))
→ id (λz. z)
→ λz. z
```

Lambda calculus — Reduction Strategies

Normal order — leftmost outermost redex is reduced first

$$\underline{\texttt{id (id (λz. id z))}}$$
$$\to \underline{\texttt{id (λz. id z)}}$$
$$\to \texttt{λz. }\underline{\texttt{id z}}$$
$$\to \texttt{λz. z}$$

Lambda calculus — Reduction Strategies

Call-by-name — like normal order
but does not reduce under λ-abstraction

$$\underline{\texttt{id (id (}\lambda\texttt{z. id z))}}$$

$$\rightarrow \underline{\texttt{id (}\lambda\texttt{z. id z)}}$$

$$\rightarrow \lambda\texttt{z. id z}$$

Lambda calculus — Reduction Strategies

Call-by-value — outermost first, arguments first, does not reduce under λ-abstraction

$$\texttt{id}\ \underline{\texttt{(id (}\lambda\texttt{z. id z))}}$$
$$\to\ \underline{\texttt{id (}\lambda\texttt{z. id z)}}$$
$$\to\ \lambda\texttt{z. id z}$$

Lambda calculus — Substitution (wrong, attempt 1)

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y && \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.\ [x \mapsto s]t_1 \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\ ([x \mapsto s]t_2)
\end{aligned}
$$

What is wrong with this definition?

Lambda calculus — Substitution (wrong, attempt 2)

$$
\begin{aligned}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} &= \mathtt{s} \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} &= \mathtt{y} && \text{if } \mathtt{y} \neq \mathtt{x} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y}.\mathtt{t}_1) &= \begin{cases} \lambda\mathtt{y}.\ \mathtt{t}_1 & \text{if } \mathtt{y} = \mathtt{x} \\ \lambda\mathtt{y}.\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 & \text{if } \mathtt{y} \neq \mathtt{x} \end{cases} \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) &= ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2)
\end{aligned}
$$

What is wrong with this definition?

Lambda calculus — Substitution (attempt 3)

$$
\begin{array}{lcll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} & = & \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} & = & \mathtt{y} & \text{if } \mathtt{y} \neq \mathtt{x} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y}.\mathtt{t}_1) & = & \begin{cases} \lambda\mathtt{y}.\ \mathtt{t}_1 & \\ \lambda\mathtt{y}.\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 & \end{cases} & \begin{array}{l} \text{if } \mathtt{y} = \mathtt{x} \\ \text{if } \mathtt{y} \neq \mathtt{x} \text{ and } \mathtt{y} \notin FV(\mathtt{s}) \end{array} \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) & = & ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) &
\end{array}
$$

What is missing in this definition?

Lambda calculus — Alpha-equivalence

$$\lambda z.\ \lambda x.\ \lambda y.\ x\ (y\ z)$$

is the alpha-equivalent to

$$\lambda a.\ \lambda b.\ \lambda c.\ b\ (c\ a)$$

Names of bound variables do not matter!

Lambda calculus

→ *(untyped)*

Syntax

$\mathtt{t}$	::=		*terms:*
		$\mathtt{x}$	*variable*
		$\lambda\mathtt{x.t}$	*abstraction*
		$\mathtt{t\ t}$	*application*
$\mathtt{v}$	::=		*values:*
		$\lambda\mathtt{x.t}$	*abstraction value*

Evaluation $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \qquad \text{(E-APP1)}$$

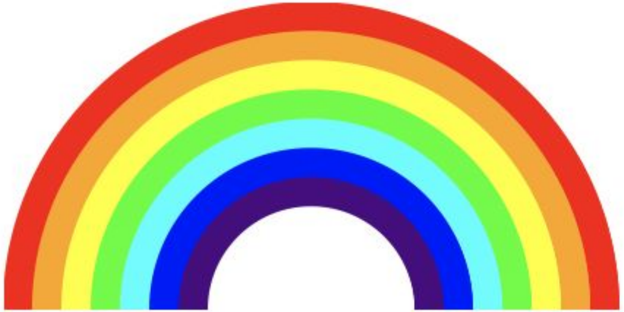
$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \text{(E-APP2)}$$

$$(\lambda\mathtt{x.t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-APPABS)}$$

Figure 5-3: Untyped lambda-calculus (λ)

Homework (self-study)

1. Install **DrRacket** https://download.racket-lang.org
2. Read **Quick: An Introduction to Racket with Pictures** https://docs.racket-lang.org/quick/index.html
3. Read about **Racket Essentials 2.1–2.2** https://docs.racket-lang.org/guide/to-scheme.html
4. Test yourself by implementing a program that renders a rainbow:



Mud cards

References

1. Lambda calculus — TaPL 5.1–5.3