TYPES OF PROGRAMMING LANGUAGES

PRINCIPLES OF PROGRAMMING LANGUAGES

Norbert Zeh

Winter 2019

Dalhousie University

REASONS TO CHOOSE A PARTICULAR PROGRAMMING LANGUAGE

REASONS TO CHOOSE A PARTICULAR PROGRAMMING LANGUAGE

- Easy to express complex ideas
- Easy to control exactly how the computation is carried out
- Rich set of data types
- Extensive (standard) library
- Active, friendly community
- Was used for this project before I joined
- Good compiler support
- Open-source

WHY ARE THERE SO MANY PROGRAMMING LANGUAGES?

WHY ARE THERE SO MANY PROGRAMMING LANGUAGES?

- Trade-off between features
  - Speed vs expressiveness/elegance
  - Speed, type safety vs rapid development
  - ...

WHY ARE THERE SO MANY PROGRAMMING LANGUAGES?

- Trade-off between features
  - Speed vs expressiveness/elegance
  - Speed, type safety vs rapid development
  - ...
- Tinkering with programming languages is fun
  ... and then they sometimes get adopted.

WHY ARE THERE SO MANY PROGRAMMING LANGUAGES?

- Trade-off between features
  - Speed vs expressiveness/elegance
  - Speed, type safety vs rapid development
  - ...
- Tinkering with programming languages is fun
  ... and then they sometimes get adopted.
- Corporate pressure

GENERAL PURPOSE VS SPECIAL PURPOSE

General-purpose programming language:

Special-purpose programming language:

General-purpose programming language:

- Designed to express arbitrary computations
- Turing-complete

Special-purpose programming language:

GENERAL PURPOSE VS SPECIAL PURPOSE

General-purpose programming language:

- Designed to express arbitrary computations
- Turing-complete

Special-purpose programming language:

- Designed to make it easy to express certain types of programs
- May not be Turing-complete

GENERAL PURPOSE VS SPECIAL PURPOSE

General-purpose programming language:

- Designed to express arbitrary computations
- Turing-complete

Special-purpose programming language:

- Designed to make it easy to express certain types of programs
- May not be Turing-complete

Examples:

GENERAL PURPOSE VS SPECIAL PURPOSE

General-purpose programming language:

- Designed to express arbitrary computations
- Turing-complete

Examples:

- C, C++, Java, Python, Ruby

Special-purpose programming language:

- Designed to make it easy to express certain types of programs
- May not be Turing-complete

GENERAL PURPOSE VS SPECIAL PURPOSE

General-purpose programming language:

- Designed to express arbitrary computations
- Turing-complete

Examples:

- C, C++, Java, Python, Ruby
- Haskell, Scheme, Prolog

Special-purpose programming language:

- Designed to make it easy to express certain types of programs
- May not be Turing-complete

GENERAL PURPOSE VS SPECIAL PURPOSE

General-purpose programming language:

- Designed to express arbitrary computations
- Turing-complete

Examples:

- C, C++, Java, Python, Ruby
- Haskell, Scheme, Prolog
- *Lua*, *Tcl/Tk*

Special-purpose programming language:

- Designed to make it easy to express certain types of programs
- May not be Turing-complete

GENERAL PURPOSE VS SPECIAL PURPOSE

General-purpose programming language:

- Designed to express arbitrary computations
- Turing-complete

Examples:

- C, C++, Java, Python, Ruby
- Haskell, Scheme, Prolog
- *Lua*, *Tcl/Tk*
- ...

Special-purpose programming language:

- Designed to make it easy to express certain types of programs
- May not be Turing-complete

GENERAL PURPOSE VS SPECIAL PURPOSE

General-purpose programming language:

- Designed to express arbitrary computations
- Turing-complete

Examples:

- C, C++, Java, Python, Ruby
- Haskell, Scheme, Prolog
- *Lua, Tcl/Tk*
- ...

Special-purpose programming language:

- Designed to make it easy to express certain types of programs
- May not be Turing-complete

Examples:

GENERAL PURPOSE VS SPECIAL PURPOSE

General-purpose programming language:

- Designed to express arbitrary computations
- Turing-complete

Examples:

- C, C++, Java, Python, Ruby
- Haskell, Scheme, Prolog
- *Lua*, *Tcl/Tk*
- ...

Special-purpose programming language:

- Designed to make it easy to express certain types of programs
- May not be Turing-complete

Examples:

- (La)TeX, HTML/XML, XSLT

GENERAL PURPOSE VS SPECIAL PURPOSE

General-purpose programming language:

- Designed to express arbitrary computations
- Turing-complete

Examples:

- C, C++, Java, Python, Ruby
- Haskell, Scheme, Prolog
- *Lua*, *Tcl/Tk*
- ...

Special-purpose programming language:

- Designed to make it easy to express certain types of programs
- May not be Turing-complete

Examples:

- (La)TeX, HTML/XML, XSLT
- R, Matlab

GENERAL PURPOSE VS SPECIAL PURPOSE

General-purpose programming language:

- Designed to express arbitrary computations
- Turing-complete

Examples:

- C, C++, Java, Python, Ruby
- Haskell, Scheme, Prolog
- *Lua*, *Tcl/Tk*
- ...

Special-purpose programming language:

- Designed to make it easy to express certain types of programs
- May not be Turing-complete

Examples:

- (La)TeX, HTML/XML, XSLT
- R, Matlab
- sed, awk

GENERAL PURPOSE VS SPECIAL PURPOSE

General-purpose programming language:

- Designed to express arbitrary computations
- Turing-complete

Examples:

- C, C++, Java, Python, Ruby
- Haskell, Scheme, Prolog
- *Lua, Tcl/Tk*
- ...

Special-purpose programming language:

- Designed to make it easy to express certain types of programs
- May not be Turing-complete

Examples:

- (La)TeX, HTML/XML, XSLT
- R, Matlab
- sed, awk
- ...

GENERAL PURPOSE VS SPECIAL PURPOSE

General-purpose programming language:

- Designed to express arbitrary computations
- Turing-complete

Examples:

- C, C++, Java, Python, Ruby
- Haskell, Scheme, Prolog
- *Lua*, *Tcl/Tk*
- ...

Special-purpose programming language:

- Designed to make it easy to express certain types of programs
- May not be Turing-complete

Examples:

- (La)TeX, HTML/XML, XSLT
- R, Matlab
- sed, awk
- ...

TYPES OF PROGRAMMING LANGUAGES

TYPES OF PROGRAMMING LANGUAGES

C++

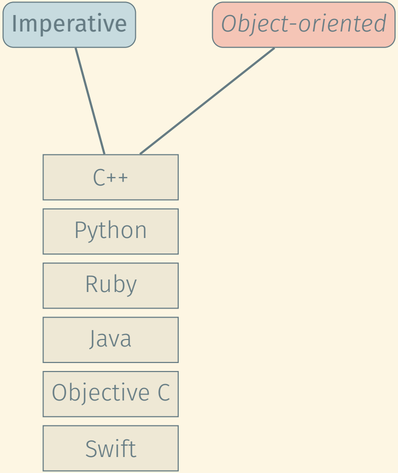
Python

Ruby

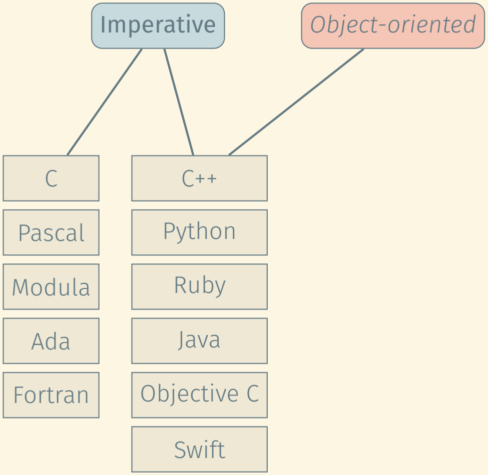
Java

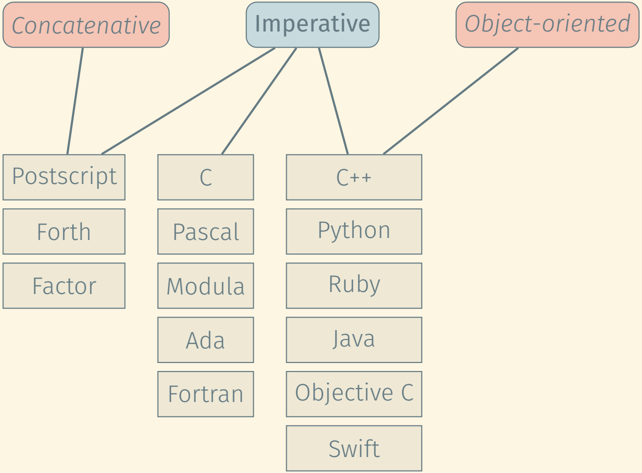
Objective C

Swift

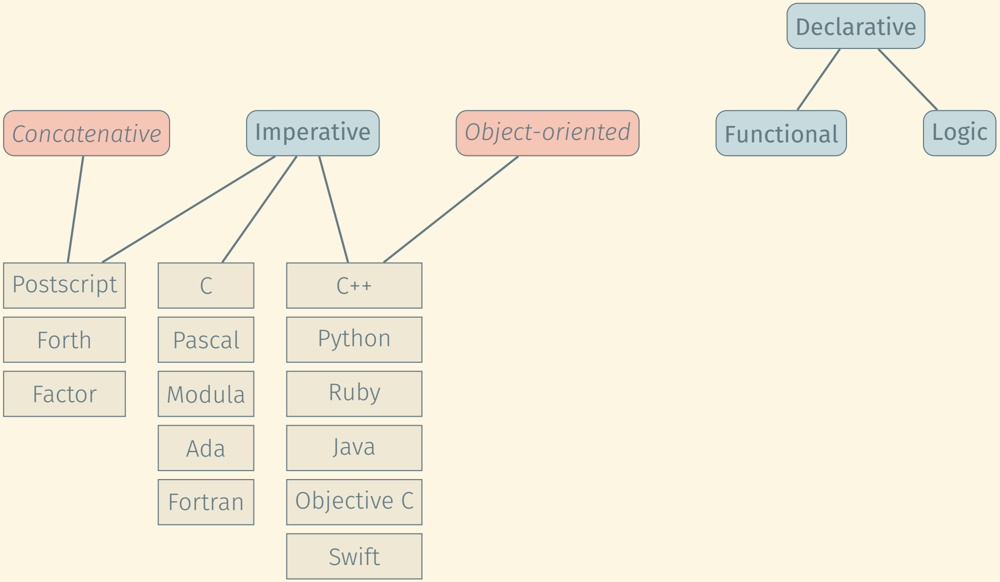

Imperative
Object-oriented
C++
Python
Ruby
Java
Objective C
Swift

TYPES OF PROGRAMMING LANGUAGES

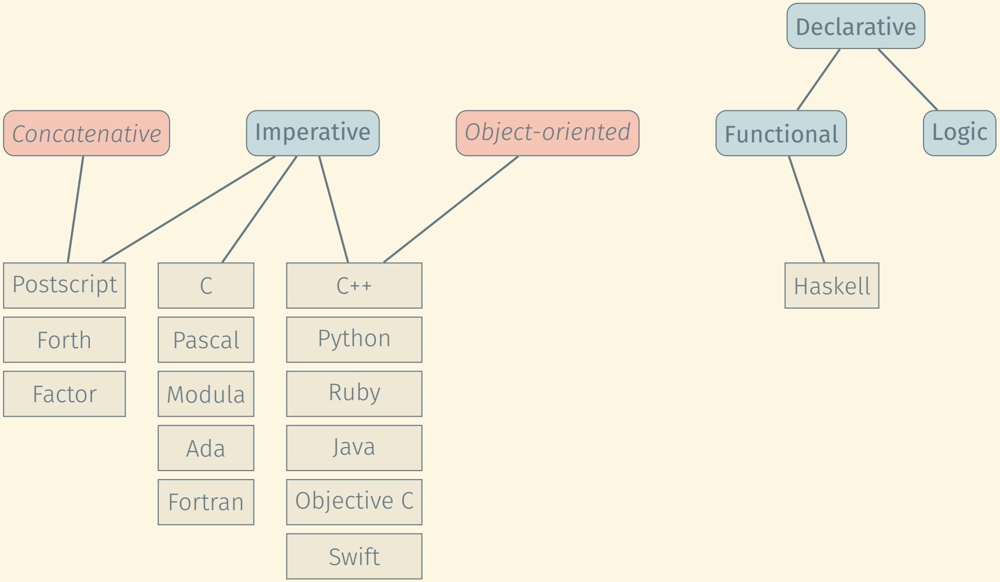


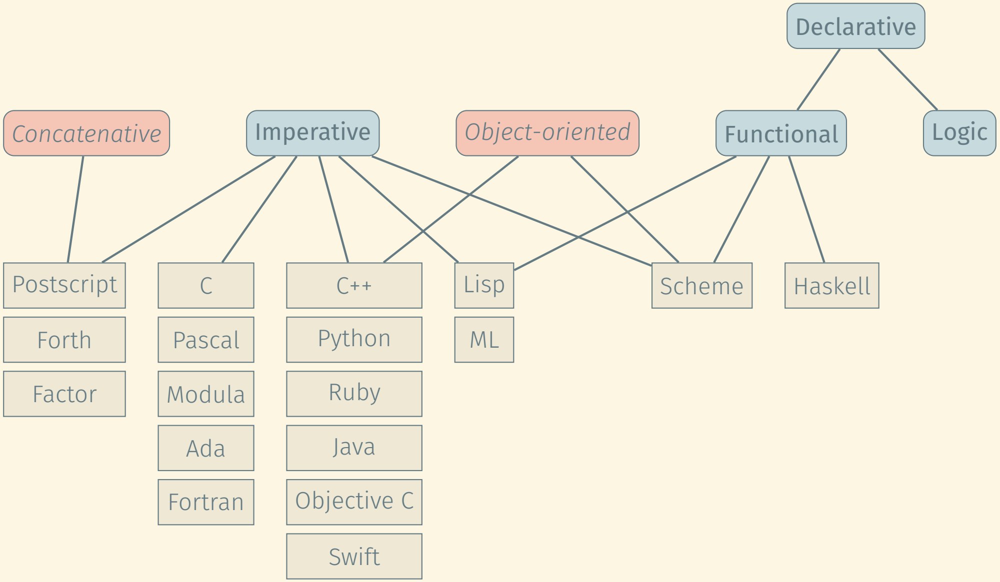

Concatenative
Imperative
Object-oriented
Postscript
Forth
Factor
C
Pascal
Modula
Ada
Fortran
C++
Python
Ruby
Java
Objective C
Swift

TYPES OF PROGRAMMING LANGUAGES

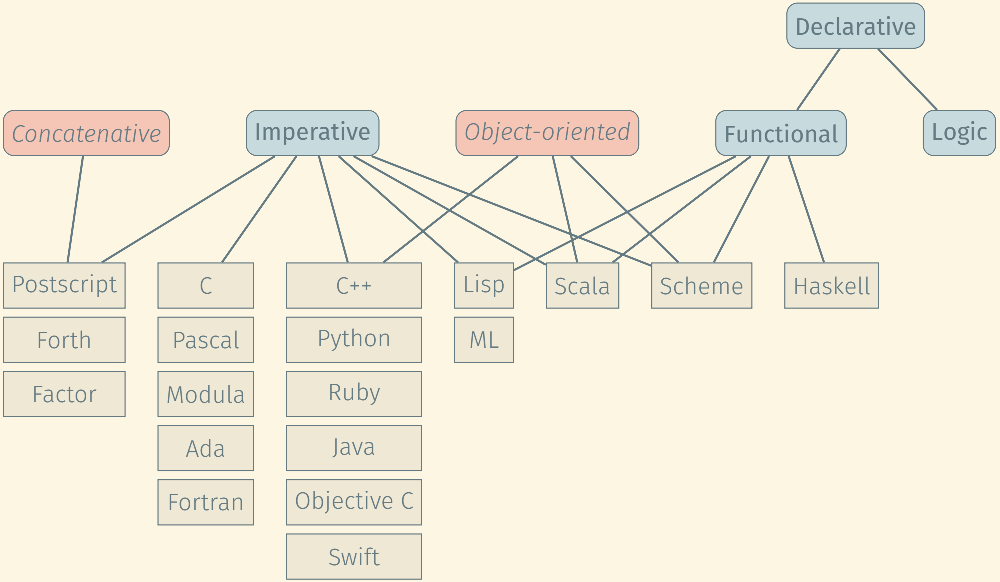


TYPES OF PROGRAMMING LANGUAGES

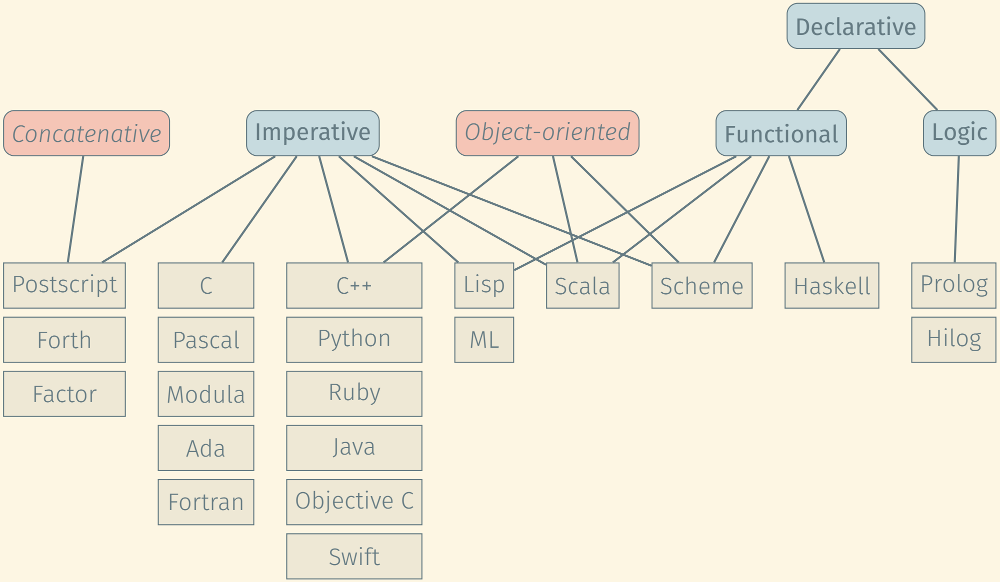



Declarative
Concatenative
Imperative
Object-oriented
Functional
Logic
Postscript
Forth
Factor
C
Pascal
Modula
Ada
Fortran
C++
Python
Ruby
Java
Objective C
Swift
Lisp
ML
Scheme
Haskell

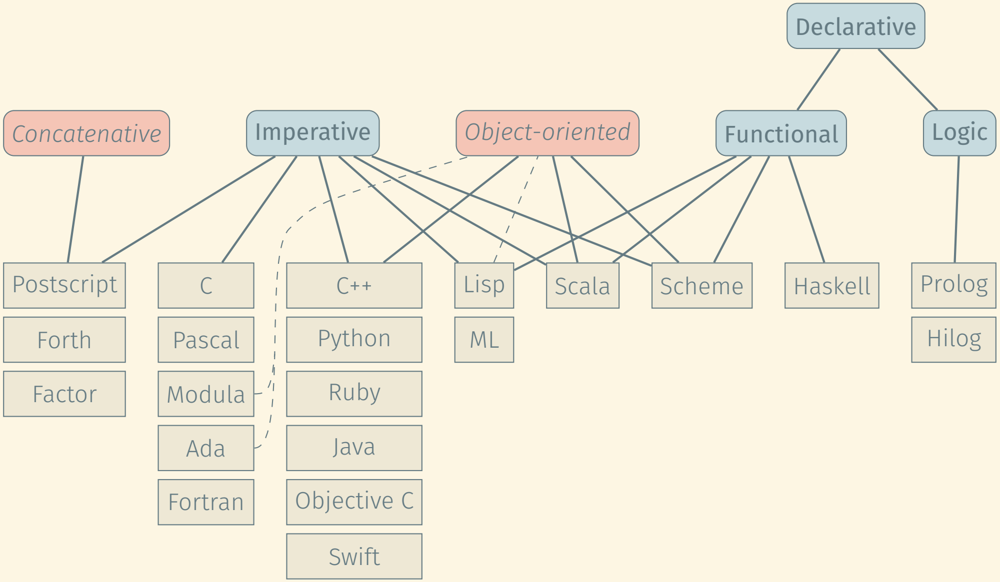
TYPES OF PROGRAMMING LANGUAGES



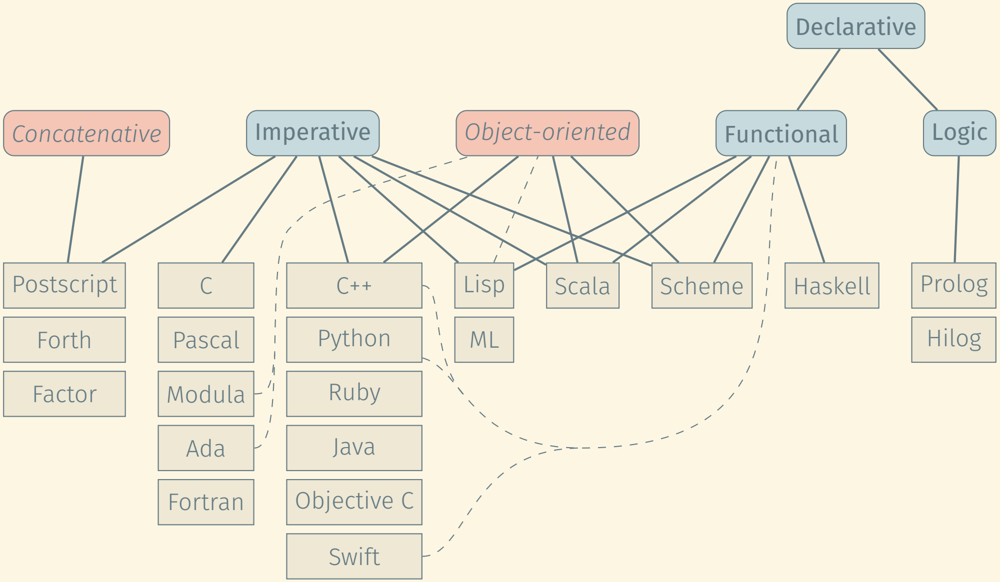
TYPES OF PROGRAMMING LANGUAGES



TYPES OF PROGRAMMING LANGUAGES



TYPES OF PROGRAMMING LANGUAGES



THE LANGUAGES I USE

Rust or C++

- When I need performance

C

- When I feel nostalgic

Haskell

- When I want to have fun and write elegant code that I trust

Prolog

- When I want to solve puzzles

Python

- When I need to write a prototype quickly

Scala

- When I'm told to use Java

Java

- Never

Scheme

- When I'd rather teach you Haskell

IMPERATIVE VS DECLARATIVE PROGRAMMING

Imperative programming:

Declarative programming:

Imperative programming:

- Focus on telling the computer exactly which steps to execute

Declarative programming:

IMPERATIVE VS DECLARATIVE PROGRAMMING

Imperative programming:

- Focus on telling the computer exactly which steps to execute

Declarative programming:

- Focus on telling the computer ***what*** to do, ***not how*** to do it

IMPERATIVE VS DECLARATIVE PROGRAMMING

Imperative programming:

- Focus on telling the computer exactly which steps to execute
- Close to the machine

Declarative programming:

- Focus on telling the computer ***what*** to do, ***not how*** to do it

IMPERATIVE VS DECLARATIVE PROGRAMMING

Imperative programming:

- Focus on telling the computer exactly which steps to execute
- Close to the machine

Declarative programming:

- Focus on telling the computer ***what*** to do, ***not how*** to do it
- More high-level, elegant, expressive

IMPERATIVE VS DECLARATIVE PROGRAMMING

Imperative programming:

- Focus on telling the computer exactly which steps to execute
- Close to the machine

Declarative programming:

- Focus on telling the computer ***what*** to do, ***not how*** to do it
- More high-level, elegant, expressive
- Need to include evaluation engine in run-time system

IMPERATIVE VS DECLARATIVE PROGRAMMING

Imperative programming:

- Focus on telling the computer exactly which steps to execute
- Close to the machine

Declarative programming:

- Focus on telling the computer ***what*** to do, ***not how*** to do it
- More high-level, elegant, expressive
- Need to include evaluation engine in run-time system
- Can come with performance penalties

IMPERATIVE VS DECLARATIVE PROGRAMMING

Imperative programming:

- Focus on telling the computer exactly which steps to execute
- Close to the machine
- Difficult to analyze/ automatically optimize

Declarative programming:

- Focus on telling the computer ***what*** to do, ***not how*** to do it
- More high-level, elegant, expressive
- Need to include evaluation engine in run-time system
- Can come with performance penalties

IMPERATIVE VS DECLARATIVE PROGRAMMING

Imperative programming:

- Focus on telling the computer exactly which steps to execute
- Close to the machine
- Difficult to analyze/automatically optimize

Declarative programming:

- Focus on telling the computer ***what*** to do, ***not how*** to do it
- More high-level, elegant, expressive
- Need to include evaluation engine in run-time system
- Can come with performance penalties
- Easier to analyze/automatically optimize/parallelize

IMPERATIVE VS DECLARATIVE PROGRAMMING

Imperative programming:

- Focus on telling the computer exactly which steps to execute
- Close to the machine
- Difficult to analyze/ automatically optimize
- Functions called for
  - Return values
  - Side effects

Declarative programming:

- Focus on telling the computer ***what*** to do, ***not how*** to do it
- More high-level, elegant, expressive
- Need to include evaluation engine in run-time system
- Can come with performance penalties
- Easier to analyze/automatically optimize/parallelize

SIDE EFFECTS

- Variable updates, I/O (disk, screen, keyboard, network, ...)

SIDE EFFECTS

- Variable updates, I/O (disk, screen, keyboard, network, …)

… are evil:

- Source of 90% of all software bugs

SIDE EFFECTS

- Variable updates, I/O (disk, screen, keyboard, network, ...)

... are evil:

- Source of 90% of all software bugs

... are the only reason we compute at all:

- Taking input and communicating results requires side effects.

FUNCTIONAL PROGRAMMING

- Functions have no side effects

FUNCTIONAL PROGRAMMING

- Functions have no side effects
- Variables are immutable once defined

FUNCTIONAL PROGRAMMING

- Functions have no side effects
- Variables are immutable once defined
- Functions are first-class objects

FUNCTIONAL PROGRAMMING

- Functions have no side effects
- Variables are immutable once defined
- Functions are first-class objects
- Can express imperative computations elegantly!

- Functions have no side effects
- Variables are immutable once defined
- Functions are first-class objects
- Can express imperative computations elegantly!

Pros:

Cons:

FUNCTIONAL PROGRAMMING

- Functions have no side effects
- Variables are immutable once defined
- Functions are first-class objects
- Can express imperative computations elegantly!

Pros:

- Easier to analyze/optimize

Cons:

- Functions have no side effects
- Variables are immutable once defined
- Functions are first-class objects
- Can express imperative computations elegantly!

Pros:

- Easier to analyze/optimize
- No specified execution order
  $\longrightarrow$ easy to parallelize

Cons:

FUNCTIONAL PROGRAMMING

- Functions have no side effects
- Variables are immutable once defined
- Functions are first-class objects
- Can express imperative computations elegantly!

Pros:

- Easier to analyze/optimize
- No specified execution order
  $\longrightarrow$ easy to parallelize

Cons:

- Can be less efficient than
  well-designed imperative code

FUNCTIONAL PROGRAMMING

- Functions have no side effects
- Variables are immutable once defined
- Functions are first-class objects
- Can express imperative computations elegantly!

Pros:

- Easier to analyze/optimize
- No specified execution order
  $\longrightarrow$ easy to parallelize

Cons:

- Can be less efficient than well-designed imperative code
- Some imperative data structures are inherently more efficient than their purely functional counterparts

General program structure:

- Database of facts, represented as logical predicates
- Rules for deducing new facts from known facts
- Execution driven by queries whether certain facts are true

General program structure:

- Database of facts, represented as logical predicates
- Rules for deducing new facts from known facts
- Execution driven by queries whether certain facts are true

Runtime system:

- Engine to perform the deduction process efficiently

General program structure:

- Database of facts, represented as logical predicates
- Rules for deducing new facts from known facts
- Execution driven by queries whether certain facts are true

Runtime system:

- Engine to perform the deduction process efficiently

Pros:

Cons:

General program structure:

- Database of facts, represented as logical predicates
- Rules for deducing new facts from known facts
- Execution driven by queries whether certain facts are true

Runtime system:

- Engine to perform the deduction process efficiently

Pros:

- Even higher abstraction than functional programming

Cons:

General program structure:

- Database of facts, represented as logical predicates
- Rules for deducing new facts from known facts
- Execution driven by queries whether certain facts are true

Runtime system:

- Engine to perform the deduction process efficiently

Pros:

- Even higher abstraction than functional programming
- In theory, no need to worry about execution details at all

Cons:

General program structure:

- Database of facts, represented as logical predicates
- Rules for deducing new facts from known facts
- Execution driven by queries whether certain facts are true

Runtime system:

- Engine to perform the deduction process efficiently

Pros:

- Even higher abstraction than functional programming
- In theory, no need to worry about execution details at all

Cons:

- In practice, need to understand execution details enough to
  - Avoid infinite loops in deduction
  - Obtain efficient programs

INTERMISSION: SCHEME AND PROLOG TUTORIALS

C++:

```
template <typename It>
void merge_sort(const It &begin, const It &end) {
  auto n = end - begin;
  if (n < 2)
    return;
  auto mid = begin + n / 2;
  merge_sort(begin, mid);
  merge_sort(mid, end);
  std::vector<std::iterator_traits<It>::value_type>
    left(begin, mid);
  std::vector<std::iterator_traits<It>::value_type>
    right(mid, end);
  merge(left, right, begin);
}
```

AN EXAMPLE WHERE FUNCTIONAL PROGRAMMING SHINES: MERGE SORT (2)

```
template <typename It>
void merge(
  const std::vector<std::iterator_traits<It>::value_type> &left,
  const std::vector<std::iterator_traits<It>::value_type> &right,
  It out) {
  auto l = left.begin(), r = right.begin();
  while (l != left.end() && r != right.end()) {
    if (*r < *l)
      *out++ = *r++;
    else
      *out++ = *l++;
  }
  while (l != left.end())
    *out++ = *l++;
  while (r != right.end())
    *out++ = *r++;
}
```

Haskell:

```
mergeSort :: Ord t => [t] -> [t]
mergeSort []  = []
mergeSort [x] = [x]
mergeSort xs  = merge (mergeSort ls) (mergeSort rs)
    where n        = length xs
          (ls, rs) = splitAt (n `div` 2) xs

merge :: Ord t => [t] -> [t] -> [t]
merge []         rs         = rs
merge ls         []         = ls
merge ls@(l:ls') rs@(r:rs') | r < l     = r : merge ls  rs'
                            | otherwise = l : merge ls' rs
```

AN EXAMPLE WHERE FUNCTIONAL PROGRAMMING SHINES: MERGE SORT (4)

Prolog:

```
list_sorted([], []).
list_sorted([X], [X]).
list_sorted(List, Sorted) :-
  list_left_right(List, Left, Right),
  list_sorted(Left, LeftSorted),
  list_sorted(Right, RightSorted),
  merged_left_right(Sorted, LeftSorted, RightSorted).

merged_left_right(Left, Left, []).
merged_left_right([R|Right], [], [R|Right]).
merged_left_right([L|Merged], [L|Left], [R|Right]) :-
  L #=< R, merged_left_right(Merged, Left, [R|Right]).
merged_left_right([R|Merged], [L|Left], [R|Right]) :-
  R #< L, merged_left_right(Merged, [L|Left], Right).
```

```
list_left_right(List, Left, Right) :-
  phrase(parse_half(List, Left), List, Right).

parse_half([], []) --> [].
parse_half([_], []) --> [].
parse_half([_,_|List], [L|Left]) --> [L], parse_half(List, Left).
```

Problem: *Build a permutation of the integers* $\{0, 1, \ldots, n-1\}$ *specified by indicating, for each element, after which element it is to be inserted.*

Example:

	1	2	3	4	5	6	
Input:	0	0	1	0	3	2	
Output:	0	4	2	6	1	3	5

AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (1)

Problem: *Build a permutation of the integers* $\{0, 1, \ldots, n-1\}$ *specified by indicating, for each element, after which element it is to be inserted.*

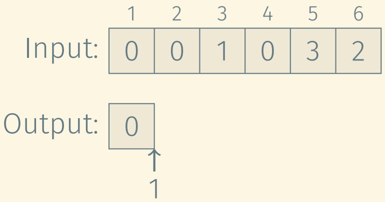
Example:

	1	2	3	4	5	6
Input:	0	0	1	0	3	2

Output:	0

Problem: *Build a permutation of the integers* $\{0, 1, \ldots, n-1\}$ *specified by indicating, for each element, after which element it is to be inserted.*

Example:



AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (1)

Problem: *Build a permutation of the integers* $\{0, 1, \ldots, n-1\}$ *specified by indicating, for each element, after which element it is to be inserted.*

Example:

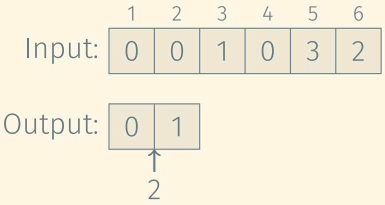
	1	2	3	4	5	6
Input:	0	0	1	0	3	2

Output:	0	1

AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (1)

Problem: *Build a permutation of the integers* $\{0, 1, \ldots, n-1\}$ *specified by indicating, for each element, after which element it is to be inserted.*

Example:



Problem: *Build a permutation of the integers* $\{0, 1, \ldots, n-1\}$ *specified by indicating, for each element, after which element it is to be inserted.*

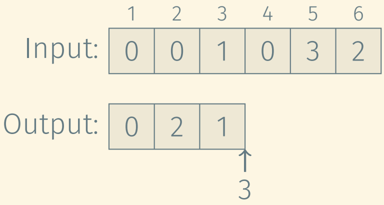
Example:

	1	2	3	4	5	6
Input:	0	0	1	0	3	2

Output:	0	2	1

Problem: *Build a permutation of the integers* $\{0, 1, \ldots, n-1\}$ *specified by indicating, for each element, after which element it is to be inserted.*

Example:



AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (1)

Problem: *Build a permutation of the integers* $\{0, 1, \ldots, n-1\}$ *specified by indicating, for each element, after which element it is to be inserted.*

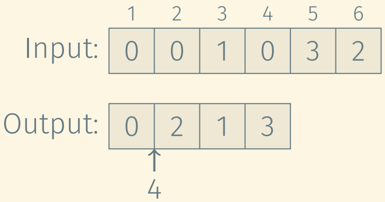
Example:

	1	2	3	4	5	6
Input:	0	0	1	0	3	2

Output:	0	2	1	3

Problem: *Build a permutation of the integers* $\{0, 1, \ldots, n-1\}$ *specified by indicating, for each element, after which element it is to be inserted.*

Example:



AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (1)

Problem: *Build a permutation of the integers* $\{0, 1, \ldots, n-1\}$ *specified by indicating, for each element, after which element it is to be inserted.*

Example:

Input:

1	2	3	4	5	6
0	0	1	0	3	2

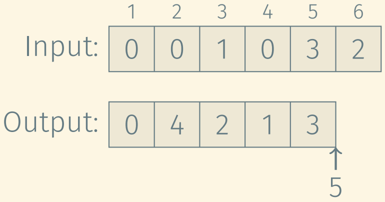
Output:

0	4	2	1	3

AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (1)

Problem: *Build a permutation of the integers* $\{0, 1, \ldots, n-1\}$ *specified by indicating, for each element, after which element it is to be inserted.*

Example:



AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (1)

Problem: *Build a permutation of the integers* $\{0, 1, \ldots, n-1\}$ *specified by indicating, for each element, after which element it is to be inserted.*

Example:

	1	2	3	4	5	6
Input:	0	0	1	0	3	2
Output:	0	4	2	1	3	5

Problem: *Build a permutation of the integers* $\{0, 1, \ldots, n-1\}$ *specified by indicating, for each element, after which element it is to be inserted.*

Example:

	1	2	3	4	5	6
Input:	0	0	1	0	3	2

Output:	0	4	2	1	3	5

↑
6

AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (1)

Problem: *Build a permutation of the integers* $\{0, 1, \ldots, n-1\}$ *specified by indicating, for each element, after which element it is to be inserted.*

Example:

	1	2	3	4	5	6
Input:	0	0	1	0	3	2

Output:	0	4	2	6	1	3	5

AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (2)

C++: Linear time

```
std::vector<int> dynamic_permute(const std::vector<int> &refs) {
  int n = ref.size() + 1;
  std::list<int> seq;
  std::vector<std::list<int>::iterator> list_nodes(n);
  list_nodes[0] = seq.insert(seq.end(), 0);
  for (int i = 1; i < n; ++i)
    list_nodes[i] = seq.insert(next(list_nodes[ref[i]]), i);
  return std::vector<int>(seq.begin(), seq.end());
}
```

AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (2)

C++: Linear time

```
std::vector<int> dynamic_permute(const std::vector<int> &refs) {
  int n = ref.size() + 1;
  std::list<int> seq;
  std::vector<std::list<int>::iterator> list_nodes(n);
  list_nodes[0] = seq.insert(seq.end(), 0);
  for (int i = 1; i < n; ++i)
    list_nodes[i] = seq.insert(next(list_nodes[ref[i]]), i);
  return std::vector<int>(seq.begin(), seq.end());
}
```

0

AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (2)

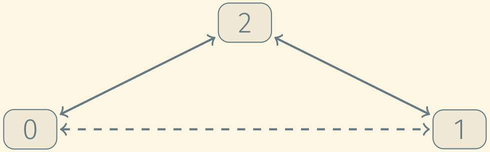
C++: Linear time

```
std::vector<int> dynamic_permute(const std::vector<int> &refs) {
  int n = ref.size() + 1;
  std::list<int> seq;
  std::vector<std::list<int>::iterator> list_nodes(n);
  list_nodes[0] = seq.insert(seq.end(), 0);
  for (int i = 1; i < n; ++i)
    list_nodes[i] = seq.insert(next(list_nodes[ref[i]]), i);
  return std::vector<int>(seq.begin(), seq.end());
}
```

AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (2)

C++: Linear time

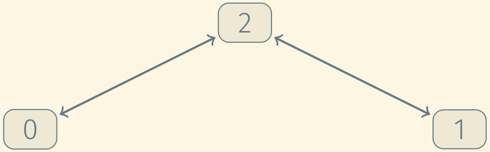
```
std::vector<int> dynamic_permute(const std::vector<int> &refs) {
  int n = ref.size() + 1;
  std::list<int> seq;
  std::vector<std::list<int>::iterator> list_nodes(n);
  list_nodes[0] = seq.insert(seq.end(), 0);
  for (int i = 1; i < n; ++i)
    list_nodes[i] = seq.insert(next(list_nodes[ref[i]]), i);
  return std::vector<int>(seq.begin(), seq.end());
}
```



AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (2)

C++: Linear time

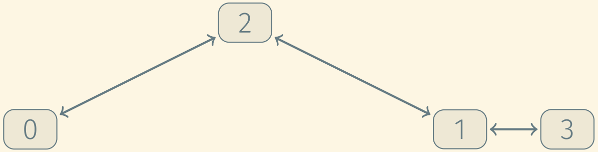
```
std::vector<int> dynamic_permute(const std::vector<int> &refs) {
  int n = ref.size() + 1;
  std::list<int> seq;
  std::vector<std::list<int>::iterator> list_nodes(n);
  list_nodes[0] = seq.insert(seq.end(), 0);
  for (int i = 1; i < n; ++i)
    list_nodes[i] = seq.insert(next(list_nodes[ref[i]]), i);
  return std::vector<int>(seq.begin(), seq.end());
}
```



AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (2)

C++: Linear time

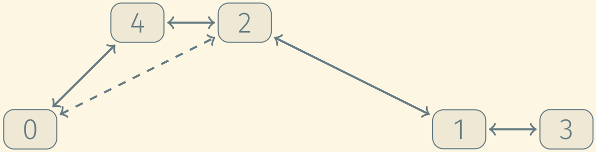
```
std::vector<int> dynamic_permute(const std::vector<int> &refs) {
  int n = ref.size() + 1;
  std::list<int> seq;
  std::vector<std::list<int>::iterator> list_nodes(n);
  list_nodes[0] = seq.insert(seq.end(), 0);
  for (int i = 1; i < n; ++i)
    list_nodes[i] = seq.insert(next(list_nodes[ref[i]]), i);
  return std::vector<int>(seq.begin(), seq.end());
}
```



AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (2)

C++: Linear time

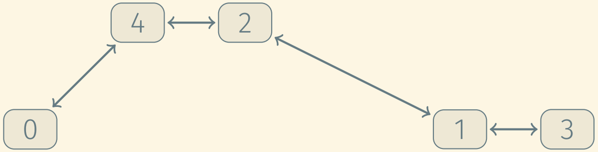
```
std::vector<int> dynamic_permute(const std::vector<int> &refs) {
  int n = ref.size() + 1;
  std::list<int> seq;
  std::vector<std::list<int>::iterator> list_nodes(n);
  list_nodes[0] = seq.insert(seq.end(), 0);
  for (int i = 1; i < n; ++i)
    list_nodes[i] = seq.insert(next(list_nodes[ref[i]]), i);
  return std::vector<int>(seq.begin(), seq.end());
}
```



AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (2)

C++: Linear time

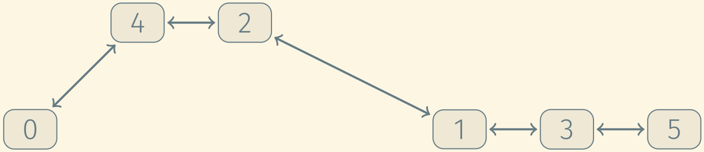
```
std::vector<int> dynamic_permute(const std::vector<int> &refs) {
  int n = ref.size() + 1;
  std::list<int> seq;
  std::vector<std::list<int>::iterator> list_nodes(n);
  list_nodes[0] = seq.insert(seq.end(), 0);
  for (int i = 1; i < n; ++i)
    list_nodes[i] = seq.insert(next(list_nodes[ref[i]]), i);
  return std::vector<int>(seq.begin(), seq.end());
}
```



AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (2)

C++: Linear time

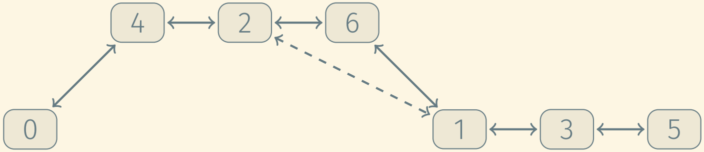
```
std::vector<int> dynamic_permute(const std::vector<int> &refs) {
  int n = ref.size() + 1;
  std::list<int> seq;
  std::vector<std::list<int>::iterator> list_nodes(n);
  list_nodes[0] = seq.insert(seq.end(), 0);
  for (int i = 1; i < n; ++i)
    list_nodes[i] = seq.insert(next(list_nodes[ref[i]]), i);
  return std::vector<int>(seq.begin(), seq.end());
}
```



AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (2)

C++: Linear time

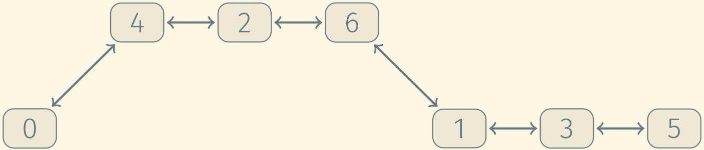
```
std::vector<int> dynamic_permute(const std::vector<int> &refs) {
  int n = ref.size() + 1;
  std::list<int> seq;
  std::vector<std::list<int>::iterator> list_nodes(n);
  list_nodes[0] = seq.insert(seq.end(), 0);
  for (int i = 1; i < n; ++i)
    list_nodes[i] = seq.insert(next(list_nodes[ref[i]]), i);
  return std::vector<int>(seq.begin(), seq.end());
}
```



AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (2)

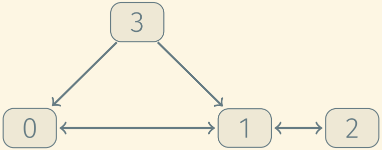
C++: Linear time

```
std::vector<int> dynamic_permute(const std::vector<int> &refs) {
  int n = ref.size() + 1;
  std::list<int> seq;
  std::vector<std::list<int>::iterator> list_nodes(n);
  list_nodes[0] = seq.insert(seq.end(), 0);
  for (int i = 1; i < n; ++i)
    list_nodes[i] = seq.insert(next(list_nodes[ref[i]]), i);
  return std::vector<int>(seq.begin(), seq.end());
}
```



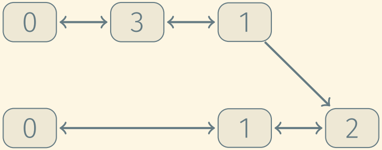
Doing this without mutation:

Doing this without mutation:



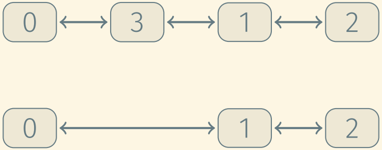
AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (3)

Doing this without mutation:

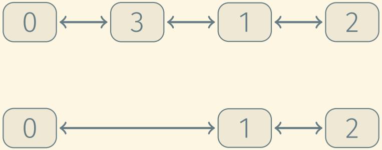


AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (3)

Doing this without mutation:



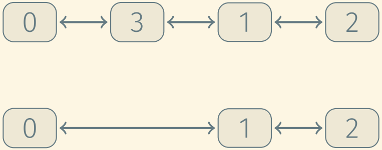
Doing this without mutation:



When “updating” any node in a functional data structure, all nodes with a path of pointers to it need to be replaced too.

AN EXAMPLE WHERE IMPERATIVE PROGRAMMING SHINES: DYNAMIC DATA STRUCTURES (3)

Doing this without mutation:



When "updating" any node in a functional data structure, all nodes with a path of pointers to it need to be replaced too.

This makes standard pointer-based data structures difficult/impossible to implement functionally.

WHAT ABOUT TRADITIONAL PERMUTING? (1)

Problem: *Given a list of elements, each annotated with its desired position in the output list, build an array storing each element in the desired position.*

Example:

Input:	(2, a)	(0, b)	(3, c)	(1, e)	(4, d)

Output:	b	e	a	c	d

WHAT ABOUT TRADITIONAL PERMUTING? (2)

C++:

```
template <typename T>
std::vector<T> permute(
  const std::vector<std::pair<int, T>> &input) {
  std::vector<T> output(input.size());
  for (auto &item : input)
    output[item.first] = item.second;
  return output;
}
```

C++:

```
template <typename T>
std::vector<T> permute(
  const std::vector<std::pair<int, T>> &input) {
  std::vector<T> output(input.size());
  for (auto &item : input)
    output[item.first] = item.second;
  return output;
}
```

Haskell:

```
permute :: [(Int, t)] -> [t]
permute xs = elems (array (0, len xs - 1) xs)
```

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (1)

4				2		6	1	
2	3		6		1	7		
	6	9	5			4		3
	9							4
	4	2				8	3	
8							9	
9		8			2	3	7	
		4	1		3		6	8
	1	6		7				2

4	8	7	3	2	9	6	1	5
2	3	5	6	4	1	7	8	9
1	6	9	5	8	7	4	2	3
6	9	1	7	3	8	2	5	4
5	4	2	9	1	6	8	3	7
8	7	3	2	5	4	1	9	6
9	5	8	4	6	2	3	7	1
7	2	4	1	9	3	5	6	8
3	1	6	8	7	5	9	4	2

Sudoku

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (2)

Prolog:

```
sudoku(Rows) :-
  transpose(Rows, Columns),
  rows_blocks(Rows, Blocks),
  append([Rows, Columns, Blocks], Sets),
  maplist(permutation([1, 2, 3, 4, 5, 6, 7, 8, 9]), Sets).
```

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (2)

Prolog:

```
sudoku(Rows) :-
  transpose(Rows, Columns),
  rows_blocks(Rows, Blocks),
  append([Rows, Columns, Blocks], Sets),
  maplist(permutation([1, 2, 3, 4, 5, 6, 7, 8, 9]), Sets).

rows_blocks([], []).
rows_blocks([R1,R2,R3|Rows], [B1,B2,B3|Blocks]) :-
  rows3_blocks3([R1,R2,R3], [B1,B2,B3]).
```

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (2)

Prolog:

```
sudoku(Rows) :-
  transpose(Rows, Columns),
  rows_blocks(Rows, Blocks),
  append([Rows, Columns, Blocks], Sets),
  maplist(permutation([1, 2, 3, 4, 5, 6, 7, 8, 9]), Sets).

rows_blocks([], []).
rows_blocks([R1,R2,R3|Rows], [B1,B2,B3|Blocks]) :-
  rows3_blocks3([R1,R2,R3], [B1,B2,B3]).

rows3_blocks3([[R11,R12,R13|R1], [R21,R22,R23|R2], [R31,R32,R33|R3]],
              [[R11,R12,R13,R21,R22,R23,R31,R32,R33|Bs]) :-
  rows3_blocks3([R1,R2,R3], Bs).
```

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (2)

Prolog: Elegant but (ridiculously) slow

```
sudoku(Rows) :-
  transpose(Rows, Columns),
  rows_blocks(Rows, Blocks),
  append([Rows, Columns, Blocks], Sets),
  maplist(permutation([1, 2, 3, 4, 5, 6, 7, 8, 9]), Sets).

rows_blocks([], []).
rows_blocks([R1,R2,R3|Rows], [B1,B2,B3|Blocks]) :-
  rows3_blocks3([R1,R2,R3], [B1,B2,B3]).

rows3_blocks3([[R11,R12,R13|R1], [R21,R22,R23|R2], [R31,R32,R33|R3]],
              [[R11,R12,R13,R21,R22,R23,R31,R32,R33|Bs]) :-
  rows3_blocks3([R1,R2,R3], Bs).
```

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (3)

Prolog: Elegant and fast

```
sudoku(Rows) :-
  transpose(Rows, Columns),
  append(Rows, Vs), Vs ins 1..9,
  maplist(all_distinct, Rows),
  maplist(all_distinct, Columns),
  Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
  blocks(As,Bs,Cs), blocks(Ds,Es,Fs), blocks(Gs,Hs,Is),
  label(Vs).
```

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (3)

Prolog: Elegant and fast

```
sudoku(Rows) :-
  transpose(Rows, Columns),
  append(Rows, Vs), Vs ins 1..9,
  maplist(all_distinct, Rows),
  maplist(all_distinct, Columns),
  Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
  blocks(As,Bs,Cs), blocks(Ds,Es,Fs), blocks(Gs,Hs,Is),
  label(Vs).
```

```
blocks([], [], []).
blocks([A1,A2,A3|As], [B1,B2,B3|Bs], [C1,C2,C3|Cs]) :-
  all_distinct([A1,A2,A3,B1,B2,B3,C1,C2,C3]), blocks(As,Bs,Cs).
```

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (3)

Prolog: Elegant and fast

```
sudoku(Rows) :-
  transpose(Rows, Columns),
  append(Rows, Vs), Vs ins 1..9,
  maplist(all_distinct, Rows),
  maplist(all_distinct, Columns),
  Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
  blocks(As,Bs,Cs), blocks(Ds,Es,Fs), blocks(Gs,Hs,Is),
  label(Vs).
```

```
blocks([], [], []).
blocks([A1,A2,A3|As], [B1,B2,B3|Bs], [C1,C2,C3|Cs]) :-
  all_distinct([A1,A2,A3,B1,B2,B3,C1,C2,C3]), blocks(As,Bs,Cs).
```

What's different?

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (3)

Prolog: Elegant and fast

```
sudoku(Rows) :-
  transpose(Rows, Columns),
  append(Rows, Vs), Vs ins 1..9,
  maplist(all_distinct, Rows),
  maplist(all_distinct, Columns),
  Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
  blocks(As,Bs,Cs), blocks(Ds,Es,Fs), blocks(Gs,Hs,Is),
  label(Vs).

blocks([], [], []).
blocks([A1,A2,A3|As], [B1,B2,B3|Bs], [C1,C2,C3|Cs]) :-
  all_distinct([A1,A2,A3,B1,B2,B3,C1,C2,C3]), blocks(As,Bs,Cs).
```

What's different? This uses efficient constraint propagation.

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (4)

Prolog:

- 12 LOC
- Instantaneous answer
- SWI Prolog
  - Free, well maintained, feature-rich, ISO compliant
  - Much slower than SICSTUS Prolog (commercial)

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (4)

Prolog:

- 12 LOC
- Instantaneous answer
- SWI Prolog
  - Free, well maintained, feature-rich, ISO compliant
  - Much slower than SICSTUS Prolog (commercial)

Python:

- SAT solver (250 LOC)
- Encode puzzle as CNF (100 LOC)
- Instantaneous answer
- Could get faster if
  - Implemented in C++
  - Using state-of-the-art SAT solver

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (5)

	0			1				0					
1			0		1					1	1		
			0				1						
		1											
			0				0		1		1		1
				0							1		
1		1			1				1				
1		1					1					0	
				1			1						
						0				0	0		
		1	1							0	0		1
	0	0						0					
						1							1

⟶

1	0	0	1	1	0	1	1	0	1	0	0	1	0
1	1	0	0	1	1	0	0	1	0	1	1	0	0
0	1	1	0	0	1	0	1	1	0	0	1	0	1
0	0	1	1	0	0	1	0	0	1	1	0	1	1
1	0	0	1	1	0	0	1	1	0	1	0	1	0
0	1	1	0	0	1	0	0	1	1	0	1	0	1
0	1	0	1	0	0	1	1	0	0	1	1	0	1
1	0	1	0	1	1	0	0	1	1	0	0	1	0
1	0	1	1	0	0	1	1	0	0	1	1	0	0
0	1	0	0	1	0	1	1	0	0	1	1	0	1
1	1	0	0	1	1	0	0	1	1	0	0	1	0
0	0	1	1	0	1	0	0	1	1	0	0	1	1
1	0	0	1	1	0	1	1	0	0	1	1	0	0
0	1	1	0	0	1	1	0	0	1	0	0	1	1

Binary Puzzle

- No two identical rows/columns
- #0s = #1s in each row/column
- No three consecutive 0s or 1s in any row or column

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (6)

```
binary(Rows) :-
  append(Rows, Vs), Vs ins 0..1,
  transpose(Rows, Columns),
  maplist(no_triplets, Rows),
  maplist(no_triplets, Columns),
  maplist(zero_one_balance, Rows),
  maplist(zero_one_balance, Columns),
  phrase(pairs(Rows), Row_Pairs),
  phrase(pairs(Columns), Column_Pairs),
  maplist(not_same, Row_Pairs),
  maplist(not_same, Column_Pairs),
  label(Vs).
```

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (6)

```
binary(Rows) :-
  append(Rows, Vs), Vs ins 0..1,
  transpose(Rows, Columns),
  maplist(no_triplets, Rows),
  maplist(no_triplets, Columns),
  maplist(zero_one_balance, Rows),
  maplist(zero_one_balance, Columns),
  phrase(pairs(Rows), Row_Pairs),
  phrase(pairs(Columns), Column_Pairs),
  maplist(not_same, Row_Pairs),
  maplist(not_same, Column_Pairs),
  label(Vs).

no_triplets(List) :-
  length(List,L), L < 3.
no_triplets([A,B,C|List]) :-
  A+B+C #> 0, A+B+C #< 3,
  no_triplets([B,C|List]).
```

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (6)

```
binary(Rows) :-
  append(Rows, Vs), Vs ins 0..1,
  transpose(Rows, Columns),
  maplist(no_triplets, Rows),
  maplist(no_triplets, Columns),
  maplist(zero_one_balance, Rows),
  maplist(zero_one_balance, Columns),
  phrase(pairs(Rows), Row_Pairs),
  phrase(pairs(Columns), Column_Pairs),
  maplist(not_same, Row_Pairs),
  maplist(not_same, Column_Pairs),
  label(Vs).

no_triplets(List) :-
  length(List,L), L < 3.
no_triplets([A,B,C|List]) :-
  A+B+C #> 0, A+B+C #< 3,
  no_triplets([B,C|List]).
```

```
zero_one_balance(List) :-
  length(List,L), Half is L // 2,
  sum(List, #= Half).
```

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (6)

```
binary(Rows) :-
  append(Rows, Vs), Vs ins 0..1,
  transpose(Rows, Columns),
  maplist(no_triplets, Rows),
  maplist(no_triplets, Columns),
  maplist(zero_one_balance, Rows),
  maplist(zero_one_balance, Columns),
  phrase(pairs(Rows), Row_Pairs),
  phrase(pairs(Columns), Column_Pairs),
  maplist(not_same, Row_Pairs),
  maplist(not_same, Column_Pairs),
  label(Vs).

no_triplets(List) :-
  length(List,L), L < 3.
no_triplets([A,B,C|List]) :-
  A+B+C #> 0, A+B+C #< 3,
  no_triplets([B,C|List]).
```

```
zero_one_balance(List) :-
  length(List,L), Half is L // 2,
  sum(List, #= Half).

not_same((List1,List2) :-
  maplist(diff, List1, List2, Diffs),
  sum(Diffs, #>, 0).

diff(A, B, Diff) :-
  Diff #<==> A #\= B.
```

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (6)

```
binary(Rows) :-
  append(Rows, Vs), Vs ins 0..1,
  transpose(Rows, Columns),
  maplist(no_triplets, Rows),
  maplist(no_triplets, Columns),
  maplist(zero_one_balance, Rows),
  maplist(zero_one_balance, Columns),
  phrase(pairs(Rows), Row_Pairs),
  phrase(pairs(Columns), Column_Pairs),
  maplist(not_same, Row_Pairs),
  maplist(not_same, Column_Pairs),
  label(Vs).

no_triplets(List) :-
  length(List,L), L < 3.
no_triplets([A,B,C|List]) :-
  A+B+C #> 0, A+B+C #< 3,
  no_triplets([B,C|List]).
```

```
zero_one_balance(List) :-
  length(List,L), Half is L // 2,
  sum(List, #= Half).

not_same((List1,List2) :-
  maplist(diff, List1, List2, Diffs),
  sum(Diffs, #>, 0).

diff(A, B, Diff) :-
  Diff #<==> A #\= B.

pairs([_]) --> [].
pairs([X|List]) -->
  pairs_(X, List), pairs(List).

pairs_(_, []) --> [].
pairs_(X, [Y|List]) -->
  [(X,Y)], pairs_(X,List).
```

AN EXAMPLE WHERE LOGIC PROGRAMMING SHINES: CONSTRAINT SATISFACTION (7)

Prolog:

- 31 LOC
- Around 7 secs to solve
- SWI Prolog
  - Free, well maintained, feature-rich, ISO compliant
  - Much slower than SICSTUS Prolog (commercial)

Python:

- SAT solver (250 LOC)
- Encode puzzle as CNF (150 LOC)
- Around 70 secs to solve
- Could get faster if
  - Implemented in C++
  - Using state-of-the-art SAT solver

EXPRESSIVENESS OF PROGRAMMING LANGUAGES

We want expressive programming languages that make programming easy:

- Concise
- Help to avoid common bugs
- Help to express exactly what we want

EXPRESSIVENESS OF PROGRAMMING LANGUAGES

We want expressive programming languages that make programming easy:

- Concise
- Help to avoid common bugs
- Help to express exactly what we want

What do we want exactly?

We want expressive programming languages that make programming easy:

- Concise
- Help to avoid common bugs
- Help to express exactly what we want

What do we want exactly?

Fine-grained control ⟷ High-level abstractions

EXPRESSIVENESS OF PROGRAMMING LANGUAGES

We want expressive programming languages that make programming easy:

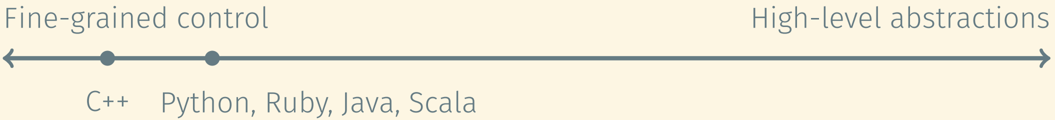
- Concise
- Help to avoid common bugs
- Help to express exactly what we want

What do we want exactly?

We want expressive programming languages that make programming easy:

- Concise
- Help to avoid common bugs
- Help to express exactly what we want

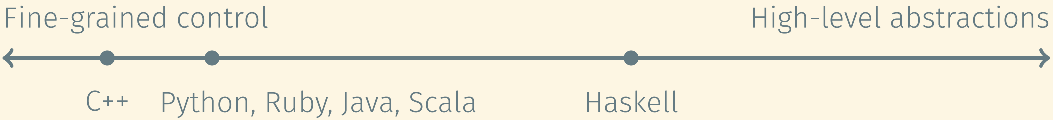
What do we want exactly?



We want expressive programming languages that make programming easy:

- Concise
- Help to avoid common bugs
- Help to express exactly what we want

What do we want exactly?

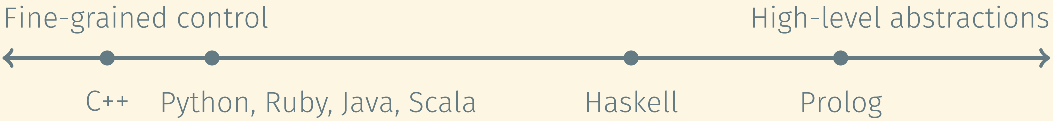


EXPRESSIVENESS OF PROGRAMMING LANGUAGES

We want expressive programming languages that make programming easy:

- Concise
- Help to avoid common bugs
- Help to express exactly what we want

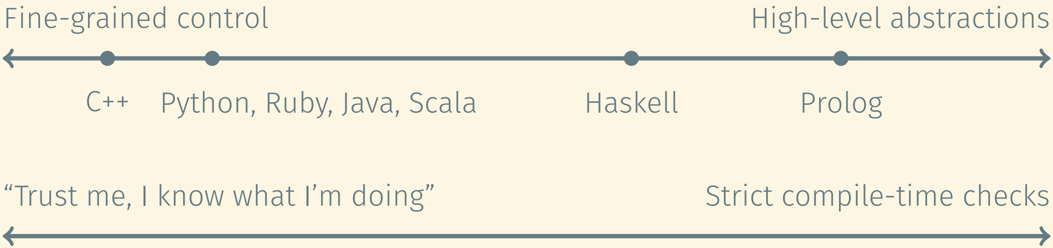
What do we want exactly?



We want expressive programming languages that make programming easy:

- Concise
- Help to avoid common bugs
- Help to express exactly what we want

What do we want exactly?

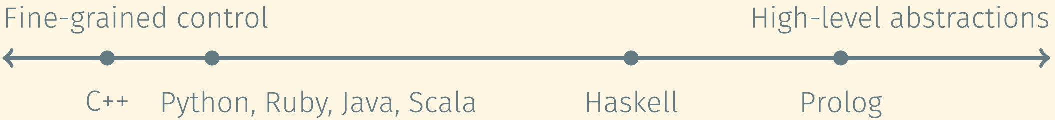


EXPRESSIVENESS OF PROGRAMMING LANGUAGES

We want expressive programming languages that make programming easy:

- Concise
- Help to avoid common bugs
- Help to express exactly what we want

What do we want exactly?

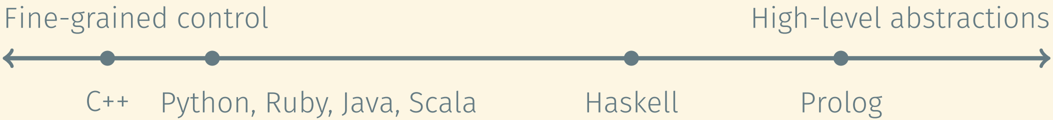


EXPRESSIVENESS OF PROGRAMMING LANGUAGES

We want expressive programming languages that make programming easy:

- Concise
- Help to avoid common bugs
- Help to express exactly what we want

What do we want exactly?

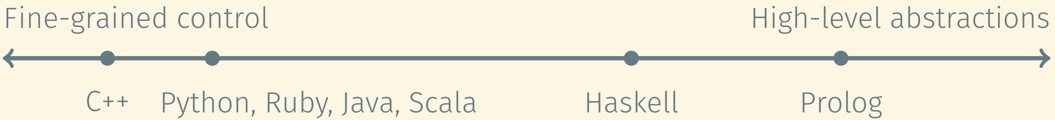


EXPRESSIVENESS OF PROGRAMMING LANGUAGES

We want expressive programming languages that make programming easy:

- Concise
- Help to avoid common bugs
- Help to express exactly what we want

What do we want exactly?

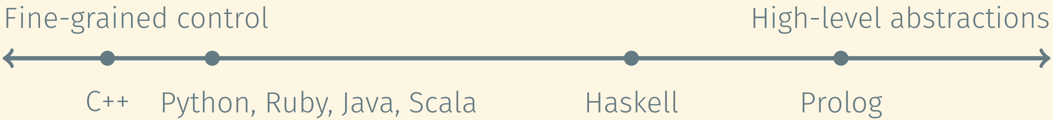


EXPRESSIVENESS OF PROGRAMMING LANGUAGES

We want expressive programming languages that make programming easy:

- Concise
- Help to avoid common bugs
- Help to express exactly what we want

What do we want exactly?



SUMMARY

Programming languages are the tools we use to express computations.

SUMMARY

Programming languages are the tools we use to express computations.

Different programming languages may be better for different jobs.

SUMMARY

Programming languages are the tools we use to express computations.

Different programming languages may be better for different jobs.

Be eager to explore new programming languages!

- Outside your comfort zone!
- It's fun.
- It makes you a better programmer, even in your favourite language.
- Your favourite language may change.