CENG 2010 - Programming Language Concepts

Week 0: Preliminaries

Burak Ekici

February 27, 2023

Table of Contents

Logistics

lecturer	Burak Ekici (burakekici@mu.edu.tr)

Logistics

lecturer	Burak Ekici (burakekici@mu.edu.tr)
teaching assistant	Erdem Türk (erdemturk@mu.edu.tr)

Logistics

lecturer	Burak Ekici (burakekici@mu.edu.tr)
teaching assistant	Erdem Türk (erdemturk@mu.edu.tr)
communication media	e-mail via MU servers – perform frequent checks

Logistics

lecturer	Burak Ekici (burakekici@mu.edu.tr)
teaching assistant	Erdem Türk (erdemturk@mu.edu.tr)
communication media	e-mail via MU servers – perform frequent checks
consultation	drop me a line anytime you want during the online lecturing period

Logistics

lecturer	Burak Ekici (burakekici@mu.edu.tr)
teaching assistant	Erdem Türk (erdemturk@mu.edu.tr)
communication media	e-mail via MU servers – perform frequent checks
consultation	drop me a line anytime you want during the online lecturing period

Important

Let me know asap in case you are in electronic equipment lack

About the Course

- Prerequisites:

About the Course

- Prerequisites:
  - Strong motivation

About the Course

- Prerequisites:
  - Strong motivation
- Resources:

About the Course

- Prerequisites:
  - Strong motivation
- Resources:
  - Introduction to Objective Caml, Jason Hickey

About the Course

- Prerequisites:
  - Strong motivation
- Resources:
  - Introduction to Objective Caml, Jason Hickey
  - Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, Yves Bertot, Pierre Castéran

About the Course

- Prerequisites:
  - Strong motivation
- Resources:
  - Introduction to Objective Caml, Jason Hickey
  - Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, Yves Bertot, Pierre Castéran
  - slides + codes

About the Course

- Prerequisites:
  - Strong motivation
- Resources:
  - Introduction to Objective Caml, Jason Hickey
  - Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, Yves Bertot, Pierre Castéran
  - slides + codes
- Grading:

Homeworks	Midterm	Final
25%	25%	50%

Rules and Academic Integrity

- Use lecture notes as your text

Rules and Academic Integrity

- Use lecture notes as your text
  - Supplement with readings, Internet

Rules and Academic Integrity

- Use lecture notes as your text
  - Supplement with readings, Internet
  - You will be responsible for everything in the notes, even if it is not directly covered in class!

Rules and Academic Integrity

- Use lecture notes as your text
  - Supplement with readings, Internet
  - You will be responsible for everything in the notes, even if it is not directly covered in class!
- All work related to the course must be done on your own

Rules and Academic Integrity

- Use lecture notes as your text
  - Supplement with readings, Internet
  - You will be responsible for everything in the notes, even if it is not directly covered in class!
- All work related to the course must be done on your own
  - Do not copy code from other students

Rules and Academic Integrity

- Use lecture notes as your text
  - Supplement with readings, Internet
  - You will be responsible for everything in the notes, even if it is not directly covered in class!
- All work related to the course must be done on your own
  - Do not copy code from other students
  - Do not copy code from the web

Rules and Academic Integrity

- Use lecture notes as your text
  - Supplement with readings, Internet
  - You will be responsible for everything in the notes, even if it is not directly covered in class!
- All work related to the course must be done on your own
  - Do not copy code from other students
  - Do not copy code from the web
  - Do not post your code on the web

Syllabus & Tentative Schedule

Week 0	Introduction and Preliminaries

Syllabus & Tentative Schedule

Week 0	Introduction and Preliminaries
Week 1	A Short Introduction to OCaml

Syllabus & Tentative Schedule

Week 0	Introduction and Preliminaries
Week 1	A Short Introduction to OCaml
Week 2	Regular Expressions and Lexical Analysis

Syllabus & Tentative Schedule

Week 0	Introduction and Preliminaries
Week 1	A Short Introduction to OCaml
Week 2	Regular Expressions and Lexical Analysis
Week 3	Context-Free Grammars and Syntactical Analysis

Syllabus & Tentative Schedule

Week 0	Introduction and Preliminaries
Week 1	A Short Introduction to OCaml
Week 2	Regular Expressions and Lexical Analysis
Week 3	Context-Free Grammars and Syntactical Analysis
Week 4	(Operational) Semantics

Syllabus & Tentative Schedule

Week 0	Introduction and Preliminaries
Week 1	A Short Introduction to OCaml
Week 2	Regular Expressions and Lexical Analysis
Week 3	Context-Free Grammars and Syntactical Analysis
Week 4	(Operational) Semantics
Week 5	(Untyped) λ-Calculus
Week 6	(Untyped) λ-Calculus

Syllabus & Tentative Schedule

Week 0	Introduction and Preliminaries
Week 1	A Short Introduction to OCaml
Week 2	Regular Expressions and Lexical Analysis
Week 3	Context-Free Grammars and Syntactical Analysis
Week 4	(Operational) Semantics
Week 5	(Untyped) λ-Calculus
Week 6	(Untyped) λ-Calculus
Week 7	Midterm

Syllabus & Tentative Schedule

Week 0	Introduction and Preliminaries
Week 1	A Short Introduction to OCaml
Week 2	Regular Expressions and Lexical Analysis
Week 3	Context-Free Grammars and Syntactical Analysis
Week 4	(Operational) Semantics
Week 5	(Untyped) λ-Calculus
Week 6	(Untyped) λ-Calculus
Week 7	Midterm
Week 8	Simply λ-Calculus
Week 9	Simply λ-Calculus

Syllabus & Tentative Schedule

Week 0	Introduction and Preliminaries
Week 1	A Short Introduction to OCaml
Week 2	Regular Expressions and Lexical Analysis
Week 3	Context-Free Grammars and Syntactical Analysis
Week 4	(Operational) Semantics
Week 5	(Untyped) λ-Calculus
Week 6	(Untyped) λ-Calculus
Week 7	Midterm
Week 8	Simply λ-Calculus
Week 9	Simply λ-Calculus
Week 10	Dependently λ-Calculus

Syllabus & Tentative Schedule

Week 0	Introduction and Preliminaries
Week 1	A Short Introduction to OCaml
Week 2	Regular Expressions and Lexical Analysis
Week 3	Context-Free Grammars and Syntactical Analysis
Week 4	(Operational) Semantics
Week 5	(Untyped) λ-Calculus
Week 6	(Untyped) λ-Calculus
Week 7	Midterm
Week 8	Simply λ-Calculus
Week 9	Simply λ-Calculus
Week 10	Dependently λ-Calculus
Week 11	The Interactive Theorem Prover Coq
Week 12	The Interactive Theorem Prover Coq
Week 13	The Interactive Theorem Prover Coq

Syllabus & Tentative Schedule

Week 0	Introduction and Preliminaries
Week 1	A Short Introduction to OCaml
Week 2	Regular Expressions and Lexical Analysis
Week 3	Context-Free Grammars and Syntactical Analysis
Week 4	(Operational) Semantics
Week 5	(Untyped) λ-Calculus
Week 6	(Untyped) λ-Calculus
Week 7	Midterm
Week 8	Simply λ-Calculus
Week 9	Simply λ-Calculus
Week 10	Dependently λ-Calculus
Week 11	The Interactive Theorem Prover Coq
Week 12	The Interactive Theorem Prover Coq
Week 13	The Interactive Theorem Prover Coq
Week 14	Final

About the Course (Course Goals)

- Study how programming languages are implemented

About the Course (Course Goals)

- Study how programming languages are implemented
  - Basics of compilers and interpreters

About the Course (Course Goals)

- Study how programming languages are implemented
  - Basics of compilers and interpreters
- Question why are there so many programming languages

About the Course (Course Goals)

- Study how programming languages are implemented
  - Basics of compilers and interpreters
- Question why are there so many programming languages
  - Not every language is perfect for every task

About the Course (Course Goals)

- Study how programming languages are implemented
  - Basics of compilers and interpreters
- Question why are there so many programming languages
  - Not every language is perfect for every task
  - Different programming paradigms

About the Course (Course Goals)

- Study how programming languages are implemented
    - Basics of compilers and interpreters
- Question why are there so many programming languages
    - Not every language is perfect for every task
    - Different programming paradigms
- Broaden your language horizons

About the Course (Course Goals)

- Study how programming languages are implemented
  - Basics of compilers and interpreters
- Question why are there so many programming languages
  - Not every language is perfect for every task
  - Different programming paradigms
- Broaden your language horizons
  - Different programming languages

About the Course (Course Goals)

- Study how programming languages are implemented
    - Basics of compilers and interpreters
- Question why are there so many programming languages
    - Not every language is perfect for every task
    - Different programming paradigms
- Broaden your language horizons
    - Different programming languages
    - Different language features and trade-offs

About the Course (Course Goals)

- Study how programming languages are implemented
  - Basics of compilers and interpreters
- Question why are there so many programming languages
  - Not every language is perfect for every task
  - Different programming paradigms
- Broaden your language horizons
  - Different programming languages
  - Different language features and trade-offs
- Study how languages are described / specified (semantics)

About the Course (Course Goals)

- Study how programming languages are implemented
  - Basics of compilers and interpreters
- Question why are there so many programming languages
  - Not every language is perfect for every task
  - Different programming paradigms
- Broaden your language horizons
  - Different programming languages
  - Different language features and trade-offs
- Study how languages are described / specified (semantics)
  - Basics of mathematical formalism of a language

About the Course (Course Sub-Goals)

- Learn some fundamental programming language concepts

About the Course (Course Sub-Goals)

- Learn some fundamental programming language concepts
  - Automata theory: Regular expressions, Context free grammars

About the Course (Course Sub-Goals)

- Learn some fundamental programming language concepts
  - Automata theory: Regular expressions, Context free grammars
- Improve programming skills

About the Course (Course Sub-Goals)

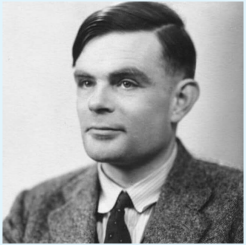
- Learn some fundamental programming language concepts
  - Automata theory: Regular expressions, Context free grammars
- Improve programming skills
  - Practice learning new programming languages: OCaml and Coq

About the Course (Course Sub-Goals)

- Learn some fundamental programming language concepts
  - Automata theory: Regular expressions, Context free grammars
- Improve programming skills
  - Practice learning new programming languages: OCaml and Coq
  - Learn how to program in two “new” styles (paradigms): functional and logical

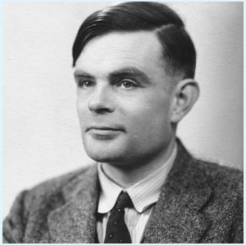
Table of Contents

Alan Turing: the father of computing



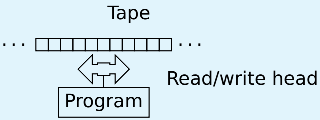
Alan Turing: the father of computing

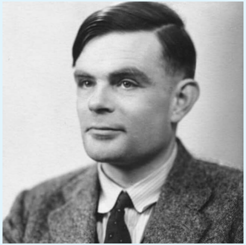
- Uncomputability: `halting problem`



Alan Turing: the father of computing

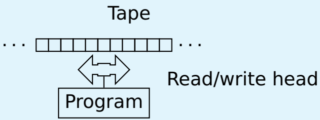
- Uncomputability: `halting problem`
- employing "*a machine*" – later named the **Turing Machine**



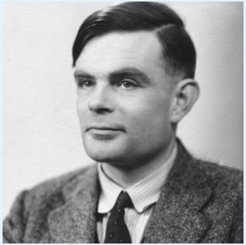


Alan Turing: the father of computing

- Uncomputability: `halting problem`
- employing “*a machine*” – later named the **Turing Machine**



- Church-Turing Thesis: computational model more expressive than Turing Machines is impossible



Equivalence of Languages

- A model, language, etc. is called Turing complete if it could simulate a Turing machine

Equivalence of Languages

- A model, language, etc. is called Turing complete if it could simulate a Turing machine
- All general purpose programming languages are roughly equivalent in the sense that they are Turing complete

  ⇒ Think as: any function that could be run using a Turing machine could be written in any of such languages

Equivalence of Languages

- A model, language, etc. is called Turing complete if it could simulate a Turing machine
- All general purpose programming languages are roughly equivalent in the sense that they are Turing complete

  ⇒ Think as: any function that could be run using a Turing machine could be written in any of such languages
- Then, is this course useless? Why do we study programming languages?

Studying Programming Languages

- Makes you program better

Studying Programming Languages

- Makes you program better
  ⇒ Features of a language makes it easier or harder to program something specific
  ⇒ Employing the right language for a specific problem makes it easier, faster and less error-prone to code
  ⇒ Ideas from one language may be used in another: many design patters in Java are indeed functional programming techniques

Studying Programming Languages

- Makes you program better
  ⇒ Features of a language makes it easier or harder to program something specific
  ⇒ Employing the right language for a specific problem makes it easier, faster and less error-prone to code
  ⇒ Ideas from one language may be used in another: many design patters in Java are indeed functional programming techniques
- Makes you learn a new language easier

Studying Programming Languages

- Makes you program better
  ⇒ Features of a language makes it easier or harder to program something specific
  ⇒ Employing the right language for a specific problem makes it easier, faster and less error-prone to code
  ⇒ Ideas from one language may be used in another: many design patters in Java are indeed functional programming techniques
- Makes you learn a new language easier
  ⇒ A programming language does not only allow you to express an idea but also shapes how you think when it comes to implement things

Language Goals

- Back in 1960s, the main goal was to compile programs so that they execute efficiently

Language Goals

- Back in 1960s, the main goal was to compile programs so that they execute efficiently
- Languages were being developed completely with respect to the hardware concepts

Language Goals

- Back in 1960s, the main goal was to compile programs so that they execute efficiently
- Languages were being developed completely with respect to the hardware concepts
- Programs had no other option than being efficient since machines were not
  ⇒ programs better be efficient today as well

Language Goals

- Back in 1960s, the main goal was to compile programs so that they execute efficiently
- Languages were being developed completely with respect to the hardware concepts
- Programs had no other option than being efficient since machines were not
  $\Rightarrow$ programs better be efficient today as well
- Machines were expensive but programming was cheap

Language Goals

- Back in 1960s, the main goal was to compile programs so that they execute efficiently
- Languages were being developed completely with respect to the hardware concepts
- Programs had no other option than being efficient since machines were not
  ⇒ programs better be efficient today as well
- Machines were expensive but programming was cheap
- Today, languages are being developed based on design concepts: inheritance, polymorphism, use of classes, functionality, ability to handle assertions, etc.

Language Goals

- Back in 1960s, the main goal was to compile programs so that they execute efficiently
- Languages were being developed completely with respect to the hardware concepts
- Programs had no other option than being efficient since machines were not
  $\Rightarrow$ programs better be efficient today as well
- Machines were expensive but programming was cheap
- Today, languages are being developed based on design concepts: inheritance, polymorphism, use of classes, functionality, ability to handle assertions, etc.
- Machines tend to be cheap but programming is expensive

Language Goals

- Back in 1960s, the main goal was to compile programs so that they execute efficiently
- Languages were being developed completely with respect to the hardware concepts
- Programs had no other option than being efficient since machines were not
  $\Rightarrow$ programs better be efficient today as well
- Machines were expensive but programming was cheap
- Today, languages are being developed based on design concepts: inheritance, polymorphism, use of classes, functionality, ability to handle assertions, etc.
- Machines tend to be cheap but programming is expensive
- Resource constraints are changing fast: power, privacy, etc.

Table of Contents

Language Features to Consider

- Syntax: what a program looks like

Language Features to Consider

- Syntax: what a program looks like
- Semantics: what a program mathematically means

Language Features to Consider

- Syntax: what a program looks like
- Semantics: what a program mathematically means
- Paradigm: how a program tends to be expressed in the language

Language Features to Consider

- Syntax: what a program looks like
- Semantics: what a program mathematically means
- Paradigm: how a program tends to be expressed in the language
- Implementation: how a program executes

Syntax

- Keywords, formatting and grammatical structure of the language

  ⇒ Usually superficial differences in between languages:

```
// in C
if (x == y) then {...} else {...}

(* in OCaml *)
if x = y then begin ... end else begin ... end
```

Syntax

- Keywords, formatting and grammatical structure of the language

  ⇒ Usually superficial differences in between languages:

```
// in C
if (x == y) then {...} else {...}

(* in OCaml *)
if x = y then begin ... end else begin ... end
```

- Concepts such as regular expressions, context-free grammars, and parsing constitute the syntax of a language

Semantics

- What does a program really mean? What does it do? – underlying meaning

	Physical Equality	Structural Equality
C	&a == &b	*a == *b
OCaml	a == b	a = b

Semantics

- What does a program really mean? What does it do? – underlying meaning

	Physical Equality	Structural Equality
C	&a == &b	*a == *b
OCaml	a == b	a = b

- Possible to be specified informally (in prose) or formally (in mathematics)

Example (Semantics of “=” vs “==” in OCaml)

```
open Printf

let num1 = ref 10

let num2 = ref 10

let () =
  let b = num1 = num2  in
  if b then printf "true" else printf "false";
  printf "\n";
  let b = num1 == num2  in
  if b then printf "true" else printf "false";
```

Example (Semantics of "==" in C)

```
#include "stdio.h"

int main (void)
{

  int *a, *b;
  int num1, num2;

  num1 = 10;
  num2 = 10;

  a = &num1;
  b = &num2;

  printf("%d\n", &a == &b);
  printf("%d\n", a == b);
  printf("%d\n", *a == *b);

  return 0;

}
```

Formal Semantics

- Mathematical definition of what programs do

Formal Semantics

- Mathematical definition of what programs do
- Imperative programs are translated into mathematical terms for property proofs

language features	$\rightarrow$	mathematical objects
types		sets or categories
terms		functions or functors
side effects		monads
...		...

Formal Semantics

- Mathematical definition of what programs do
- Imperative programs are translated into mathematical terms for property proofs

language features →	mathematical objects
types	sets or categories
terms	functions or functors
side effects	monads
...	...

- For pure programs, induction proof technique is employed. E.g., do below functions correctly compute?

```
let rec fact n = if n = 0 then 1 else n * (fact n-1)
```

```
let fact n = let rec aux i j = if i = 0 then j else aux (i-1) (j*i) in aux n 1
```

Formal Semantics

- Mathematical definition of what programs do
- Imperative programs are translated into mathematical terms for property proofs

language features →	mathematical objects
types	sets or categories
terms	functions or functors
side effects	monads
...	...

- For pure programs, induction proof technique is employed. E.g., do below functions correctly compute?

```
let rec fact n = if n = 0 then 1 else n * (fact n-1)

let fact n = let rec aux i j = if i = 0 then j else aux (i-1) (j*i) in aux n 1
```

- We will briefly consider operational (and maybe denotational) styles of semantics

Table of Contents

Paradigm

- Refers to language families

Paradigm

- Refers to language families
- There are many ways to implement and compute something

  ⇒ Some differences are superficial: for loop vs. while loop

  ⇒ Some are more fundamental:

Paradigm

- Refers to language families
- There are many ways to implement and compute something

  ⇒ Some differences are superficial: for loop vs. while loop

  ⇒ Some are more fundamental:

  - Recursion vs. looping

Paradigm

- Refers to language families
- There are many ways to implement and compute something

  ⇒ Some differences are superficial: for loop vs. while loop

  ⇒ Some are more fundamental:

  - Recursion vs. looping
  - Mutation vs. functional update

Paradigm

- Refers to language families
- There are many ways to implement and compute something

  ⇒ Some differences are superficial: for loop vs. while loop

  ⇒ Some are more fundamental:

  - Recursion vs. looping
  - Mutation vs. functional update
- Language's paradigm favors some computing methods over others

Paradigm

- Refers to language families
- There are many ways to implement and compute something

  ⇒ Some differences are superficial: for loop vs. while loop

  ⇒ Some are more fundamental:

  - Recursion vs. looping
  - Mutation vs. functional update
- Language's paradigm favors some computing methods over others
- Some cult paradigms:

Paradigm

- Refers to language families
- There are many ways to implement and compute something

  ⇒ Some differences are superficial: for loop vs. while loop

  ⇒ Some are more fundamental:
  - Recursion vs. looping
  - Mutation vs. functional update
- Language's paradigm favors some computing methods over others
- Some cult paradigms:
  - Functional

Paradigm

- Refers to language families
- There are many ways to implement and compute something

  ⇒ Some differences are superficial: for loop vs. while loop

  ⇒ Some are more fundamental:
  - Recursion vs. looping
  - Mutation vs. functional update
- Language's paradigm favors some computing methods over others
- Some cult paradigms:
  - Functional
  - Logic

Paradigm

- Refers to language families
- There are many ways to implement and compute something

  ⇒ Some differences are superficial: for loop vs. while loop

  ⇒ Some are more fundamental:

  - Recursion vs. looping
  - Mutation vs. functional update
- Language's paradigm favors some computing methods over others
- Some cult paradigms:
  - Functional
  - Logic
  - Imperative

Paradigm

- Refers to language families
- There are many ways to implement and compute something

  ⇒ Some differences are superficial: for loop vs. while loop

  ⇒ Some are more fundamental:

  - Recursion vs. looping
  - Mutation vs. functional update
- Language's paradigm favors some computing methods over others
- Some cult paradigms:
  - Functional
  - Logic
  - Imperative
  - Object-oriented

Table of Contents

Functional Languages

- Based on λ-calculus (formal system in mathematical logic)

Functional Languages

- Based on λ-calculus (formal system in mathematical logic)
- Programs are function abstractions and compositions

Functional Languages

- Based on λ-calculus (formal system in mathematical logic)
- Programs are function abstractions and compositions
- Program execution is function evaluation

Functional Languages

- Based on λ-calculus (formal system in mathematical logic)
- Programs are function abstractions and compositions
- Program execution is function evaluation
- On the same input functions always return the same output (no state effects – programs are pure)

Functional Languages

- Based on λ-calculus (formal system in mathematical logic)
- Programs are function abstractions and compositions
- Program execution is function evaluation
- On the same input functions always return the same output (no state effects – programs are pure)
- Favors immutability

Functional Languages

- Based on λ-calculus (formal system in mathematical logic)
- Programs are function abstractions and compositions
- Program execution is function evaluation
- On the same input functions always return the same output (no state effects – programs are pure)
- Favors immutability
- Functions are higher-order – could be passed as arguments, returned as results

Functional Languages

- Based on λ-calculus (formal system in mathematical logic)
- Programs are function abstractions and compositions
- Program execution is function evaluation
- On the same input functions always return the same output (no state effects – programs are pure)
- Favors immutability
- Functions are higher-order – could be passed as arguments, returned as results
- LISP (1958), ML (1973), Scheme (1975), Haskell (1987), OCaml (1987)

Basics of OCaml

- A mostly-functional, general purpose, programming language

  ⇒ Has been developed since 1987 at INRIA in France

  ⇒ Dialect of ML (1973)

  ⇒ Has objects

Basics of OCaml

- A mostly-functional, general purpose, programming language
  ⇒ Has been developed since 1987 at INRIA in France
  ⇒ Dialect of ML (1973)
  ⇒ Has objects
- Natural support for pattern matching
  ⇒ Generalizes switch/if-then-else! Very elegant!

Basics of OCaml

- A mostly-functional, general purpose, programming language
  - ⇒ Has been developed since 1987 at INRIA in France
  - ⇒ Dialect of ML (1973)
  - ⇒ Has objects
- Natural support for pattern matching
  - ⇒ Generalizes switch/if-then-else! Very elegant!
- Has full featured module system
  - ⇒ Much richer than interfaces in Java or headers in C

Basics of OCaml

- A mostly-functional, general purpose, programming language
  ⇒ Has been developed since 1987 at INRIA in France
  ⇒ Dialect of ML (1973)
  ⇒ Has objects
- Natural support for pattern matching
  ⇒ Generalizes switch/if-then-else! Very elegant!
- Has full featured module system
  ⇒ Much richer than interfaces in Java or headers in C
- Includes type inference
  ⇒ Ensures compile-time type safety

Example (Decimal to Binary in OCaml)

```
type bit =
  | Z
  | O

type binNum =
  | Bit  of bit
  | Cons of (bit * binNum)
  | Neg  of binNum
  | Pos  of binNum
```

```
let rec toBin (n: int): bitNum =
  if n <? 0 then Neg (toBin (-n))
  else if n = 0 then Bit Z
  else if n = 1 then Bit O
  else if (n mod 2 = 0) then Cons (Z, toBin(n/2))
  else Cons (O, toBin((n-1)/2))
```

Logic Programming Languages

- Also called rule-based or constraint-based

Logic Programming Languages

- Also called rule-based or constraint-based
- Allow for formalization of mathematical proofs

  $\forall \mathtt{a}\ \mathtt{b} : \mathtt{nat}, \mathtt{a} + \mathtt{b} = \mathtt{b} + \mathtt{a}$

Logic Programming Languages

- Also called rule-based or constraint-based
- Allow for formalization of mathematical proofs

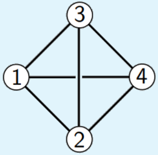
  $\forall a\, b : \mathtt{nat}, a + b = b + a$
- PROLOG (1970), Datalog (1977), Coq (1985)

Planar Graph

- A graph is called $\begin{cases} \text{complete,} & \text{if every vertex pair is connected with a unique edge} \\ \text{planar,} & \text{if it can be drawn crossing-free in the plane} \end{cases}$

Planar Graph

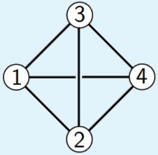
- A graph is called $\begin{cases} \text{complete,} & \text{if every vertex pair is connected with a unique edge} \\ \text{planar,} & \text{if it can be drawn crossing-free in the plane} \end{cases}$
- $\mathcal{K}_4$ – complete graph with four vertices



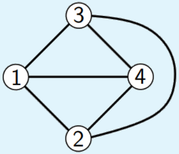
non-planar drawing

Planar Graph

- A graph is called $\begin{cases} \text{complete,} & \text{if every vertex pair is connected with a unique edge} \\ \text{planar,} & \text{if it can be drawn crossing-free in the plane} \end{cases}$
- $\mathcal{K}_4$ – complete graph with four vertices



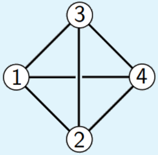
non-planar drawing



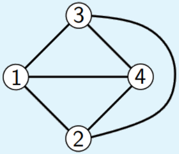
planar drawing

Planar Graph

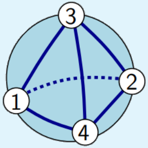
- A graph is called $\begin{cases} \text{complete,} & \text{if every vertex pair is connected with a unique edge} \\ \text{planar,} & \text{if it can be drawn crossing-free in the plane} \end{cases}$
- $\mathcal{K}_4$ – complete graph with four vertices



non-planar drawing



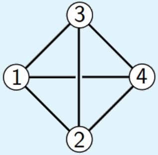
planar drawing



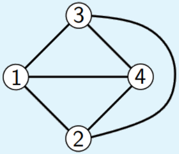
on the sphere

Planar Graph

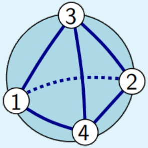
- A graph is called $\begin{cases} \text{complete,} & \text{if every vertex pair is connected with a unique edge} \\ \text{planar,} & \text{if it can be drawn crossing-free in the plane} \end{cases}$
- $\mathcal{K}_4$ – complete graph with four vertices



non-planar drawing

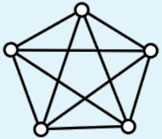


planar drawing



on the sphere

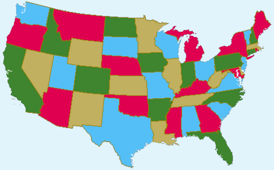
- $\mathcal{K}_5$ – complete graph with five vertices



$\mathcal{K}_n$ such that $n \geqslant 5$ has no planar drawing

The Four-Color Theorem

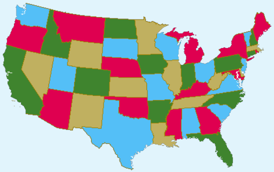
The four-color theorem states that the vertices of every planar graph can be colored with at most four colors so that no two adjacent vertices receive the same color, or for short: Every planar graph is four-colorable.



- Conjecture (Guthrie 1852)

The Four-Color Theorem

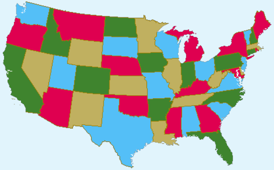
The four-color theorem states that the vertices of every planar graph can be colored with at most four colors so that no two adjacent vertices receive the same color, or for short: Every planar graph is four-colorable.



- Conjecture (Guthrie 1852)
- Proof (Kempe 1879)

The Four-Color Theorem

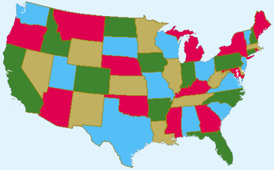
The four-color theorem states that the vertices of every planar graph can be colored with at most four colors so that no two adjacent vertices receive the same color, or for short: Every planar graph is four-colorable.



- Conjecture (Guthrie 1852)
- Proof (Kempe 1879) – Falsified (Heawood 1890)

The Four-Color Theorem

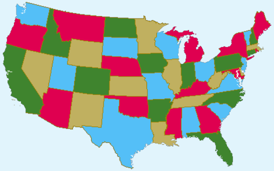
The four-color theorem states that the vertices of every planar graph can be colored with at most four colors so that no two adjacent vertices receive the same color, or for short: Every planar graph is four-colorable.



- Conjecture (Guthrie 1852)
- Proof (Kempe 1879) – Falsified (Heawood 1890)
- Another Proof (Tait 1880)

The Four-Color Theorem

The four-color theorem states that the vertices of every planar graph can be colored with at most four colors so that no two adjacent vertices receive the same color, or for short: Every planar graph is four-colorable.



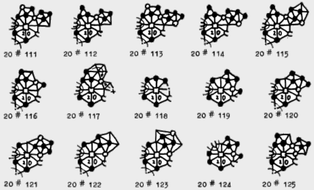
- Conjecture (Guthrie 1852)
- Proof (Kempe 1879) – Falsified (Heawood 1890)
- Another Proof (Tait 1880) – Falsified again (Petersen 1891)

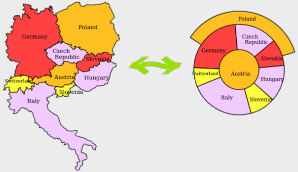
Notions

- A configuration is a collection of countries in a map

Notions

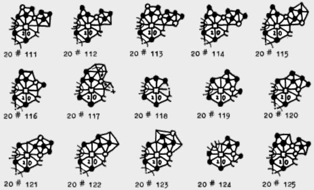
- A configuration is a collection of countries in a map
- An unavoidable set is a set of configurations such that every map contains at least one

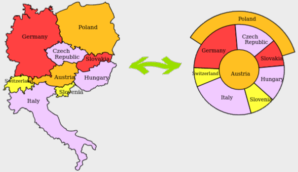




Notions

- A configuration is a collection of countries in a map
- An unavoidable set is a set of configurations such that every map contains at least one

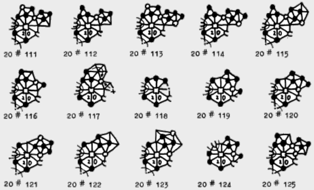


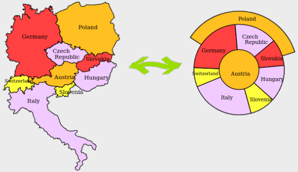


- A configuration is called reducible if any coloring of the rest of the map can be extended to include it

Notions

- A configuration is a collection of countries in a map
- An unavoidable set is a set of configurations such that every map contains at least one

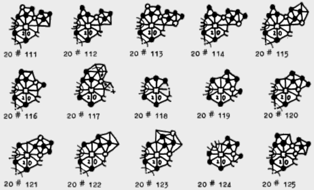


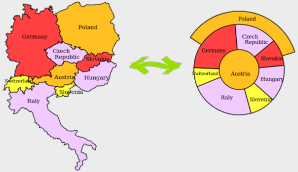


- A configuration is called reducible if any coloring of the rest of the map can be extended to include it

Notions

- A configuration is a collection of countries in a map
- An unavoidable set is a set of configurations such that every map contains at least one





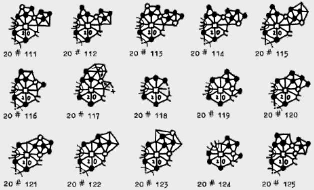
- A configuration is called reducible if any coloring of the rest of the map can be extended to include it

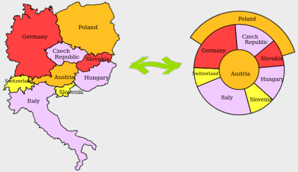
A proof idea:

- find an unavoidable set of reducible configurations

Notions

- A configuration is a collection of countries in a map
- An unavoidable set is a set of configurations such that every map contains at least one





- A configuration is called reducible if any coloring of the rest of the map can be extended to include it

A proof idea:

- find an unavoidable set of reducible configurations
- that is every map must contain at least one, and whichever it is, any coloring of the rest of the map can be extended to contain it

Proof. (The First “Correct” One)

- Appel and Haken (in 1976) solved the problem by finding an unavoidable set of 1936 (later 1478) reducible configurations
- They then used a computer program to check whether these configurations are actually reducible: if not, modify the unavoidable set
- The reaction by mathematicians

“Quelle horreur, un ordinateur !”

All Right But . . .

... are computer programs trustworthy?

What if

- Gmail would send e-mails not to expected recipients?
- your credit/debit card would withdrew without your permission?
- the auto-pilot crashes during a flight?

How on Earth!


Libération TERRE

POLITIQUE SOCIÉTÉ MONDE ÉCONOMIE CULTURE NEXT IDÉES VIDÉO PHOTO

Accueil > Terre

Un astéroïde va frôler la Terre, mais pas de panique

8 FÉVRIER 2013 À 00:23

Image fournie par la Nasa de la Terre et de la Lune. (Photo AFP)

Il s'agit du plus gros objet jamais anticipé qui s'approchera aussi près, mais il ne touchera pas la planète.

A Broad Analysis

1. User's mistake — no chance :(
2. Programmer's mistake — test
3. Compiler's mistake — test
4. Scientific mistake

A Broad Analysis

1. User's mistake — no chance :(
2. Programmer's mistake — test
3. Compiler's mistake — test
4. Scientific mistake

What is we had not done sufficient number of tests?

A Broad Analysis

1. User's mistake — no chance :(
2. Programmer's mistake — prove test
3. Compiler's mistake — prove test
4. Scientific mistake

What is we had not done sufficient number of tests?

A Broad Analysis

1. User's mistake — no chance :(
2. Programmer's mistake — prove test
3. Compiler's mistake — prove test
4. Scientific mistake

What is we had not done sufficient number of tests?
What if our proof is wrong?

A Broad Analysis

1. User's mistake — no chance :(
2. Programmer's mistake — re-evaluate prove test
3. Compiler's mistake — re-evaluate prove test
4. Scientific mistake — re-evaluate

What is we had not done sufficient number of tests?
What if our proof is wrong?

A Broad Analysis

1. User's mistake — no chance :(
2. Programmer's mistake — re-evaluate prove test
3. Compiler's mistake — re-evaluate prove test
4. Scientific mistake — re-evaluate

What is we had not done sufficient number of tests?
What if our proof is wrong?
What if we made another mistake in re-evaluating?

Motto (Verify Your Proofs!)

- Proving: hell hard!
- Verifying a proof:
  - easy if the proof is short
  - unpleasant if he proof is long

Motto (Verify Your Proofs!)

- Proving: hell hard!
- Verifying a proof:
  - easy if the proof is short
  - unpleasant if he proof is long
- Thus, we can use our computers (computational power) to verify proofs

Curry-Howard Isomorphism

Logic	$\sim$	Type Theory
Proposition		Type
Proof		Program
$\vdots$		$\vdots$

Curry-Howard Isomorphism

Logic	$\sim$	Type Theory
Proposition		Type
Proof		Program
$\vdots$		$\vdots$

- Proofs can be programmed

The Coq Proof Assistant

- A formal proof management system based on a type system – Calculus of Inductive Constructions (CIC)

The Coq Proof Assistant

- A formal proof management system based on a type system – Calculus of Inductive Constructions (CIC)
- A functional programming language (Gallina) to implement functions

The Coq Proof Assistant

- A formal proof management system based on a type system – Calculus of Inductive Constructions (CIC)
- A functional programming language (Gallina) to implement functions
- A specification language to state and prove properties of functions

The Coq Proof Assistant

- A formal proof management system based on a type system – Calculus of Inductive Constructions (CIC)
- A functional programming language (Gallina) to implement functions
- A specification language to state and prove properties of functions
- And a relatively short kernel program to check such proofs

The Chicken-Egg Problem

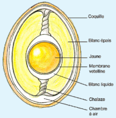
- OK., Coq verifies proofs
- But who verifies Coq?

The Chicken-Egg Problem

- OK., Coq verifies proofs
- But who verifies Coq?

Principal of de Bruijn:

- A part of Coq is weak (the part that users need to trust with no proof)
- But this part is very tiny; you can convince yourselves about that part being correct



Example (Basics of Natural Numbers)

```
Print nat.

(*
Inductive nat : Set ≜
 | O : nat
 | S : nat → nat
*)

Check S (S O).
(* 2: nat *)

Check S (S (S (S (S O)))).
(* 5: nat *)

Print Nat.add.
```

```
(*
Nat.add =
fix add (n m : nat) {struct n} : nat ≜
  match n with
    | 0 ⇒ m
    | S p ⇒ S (add p m)
  end : nat → nat → nat
*)

Locate "+".
(*
(default interpretation)
"x + y" ≜ Nat.add x y : nat_scope
*)
```

Example (Basics of Natural Numbers)

```
Print nat.

(*
Inductive nat : Set ≜
 | O : nat
 | S : nat → nat
*)

Check S (S O).
(* 2: nat *)

Check S (S (S (S (S O)))).
(* 5: nat *)

Print Nat.add.
```

```
(*
Nat.add =
fix add (n m : nat) {struct n} : nat ≜
  match n with
    | 0 ⇒ m
    | S p ⇒ S (add p m)
  end : nat → nat → nat
*)

Locate "+".
(*
(default interpretation)
"x + y" ≜ Nat.add x y : nat_scope
*)
```

```
Lemma add_comm: ∀ a b: nat, a + b = b + a.
Proof.  ...  Qed.
```

Example (Addition over natural numbers is commutative)

```
Lemma helper1: ∀ b: nat, b + 0 = b.
Proof. intro b.
        induction b; intros.
       - simpl. reflexivity.
       - cbn. rewrite IHb.
         reflexivity.
Qed.

Lemma helper2:
  ∀ a b: nat, S (a + b) = a + S b.
Proof. intro a.
        induction a; intros.
       - simpl. reflexivity.
       - simpl. rewrite (IHa b). reflexivity.
Qed.
```

```
Lemma add_comm: ∀ a b, a + b = b + a.
Proof. intro a.
        induction a; intro b.
       - simpl. rewrite (helper1 b).
         reflexivity.
       - simpl. rewrite (IHa b).
         rewrite (helper2 b a).
         reflexivity.
Qed.
```

Verified Proof of the Four-Color Theorem

In 2004 G. Gonthier produced a fully machine-checked proof of the four-color theorem using the proof assistant Coq.

- the paper: `http://www.ams.org/notices/200811/tx081101382p.pdf`
- the Coq code: `https://github.com/math-comp/fourcolor`

Table of Contents

Imperative Languages

- Also known as procedural or von Neumann languages
- Based on Turing machine
- Programs are sequence of instructions

  ⇒ Building blocks are statements, procedures and functions
- Instant contents of memory locations is in fact called the state
- The state is mutable

  ⇒ Programs that update memory locations (thus the state) are of the form (assignment – side effect)

```
int x = 10;
while (x ?> 0)
{  x = x − 1;  ... }
```

- FORTRAN (1954), Pascal (1970), C (1971)

Object-Oriented Language

- Programs are built from objects
  ⇒ Objects combine functions and data
  ⇒ Often into “classes” which can inherit

```
class C { int x; int getX() { return x; } ... }
class D extends C { ... }
```

- Smalltalk (1969), C++ (1986), Ruby (1993), Java (1995)

Scripting Languages

- To automate common tasks in a program

  $\Rightarrow$ Traditionally: text processing, extracting information from a data set, etc
- Scripting has a broad range

  $\Rightarrow$ The basis may be imperative, functional, OO, ...
- Less code intensive as compared to traditional programming languages
- sh (1971), perl (1987)

Concurrent/Parallel Languages

- Traditional languages had one thread of control
  $\Rightarrow$ Processor executes one instruction at a time
- Newer languages support many threads
  $\Rightarrow$ Thread execution conceptually independent
  $\Rightarrow$ Means to create and communicate among threads
- Concurrency may help/harm
  $\Rightarrow$ Readability, performance, expressiveness

Beyond the Paradigm

- Other important features:

  ⇒ Regular expression handling, Objects, Closures/code blocks, Immutability, Tail recursion, Pattern matching, Unification, Abstract types, Garbage collection, etc.
- Declarations

  ⇒ Explicit vs Implicit
- Type system: Static vs Dynamic, Type checking, Type safety, etc.

Table of Contents

Implementation

- How do we implement a programming language?

Implementation

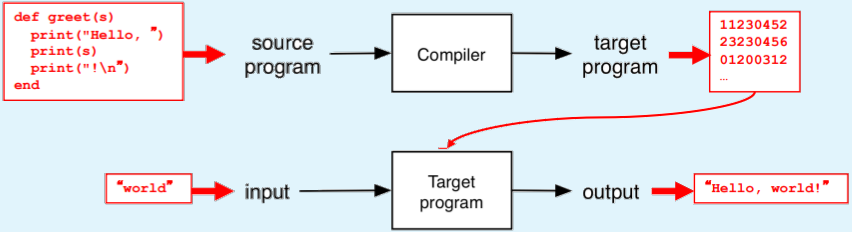
- How do we implement a programming language?
- How do we execute the program P written in some language $\mathcal{L}$?

Implementation

- How do we implement a programming language?
- How do we execute the program P written in some language $\mathcal{L}$?
- Two broad ways: Compilation and Interpretation

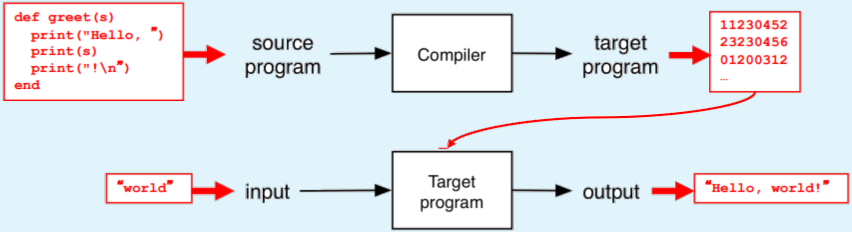
Compilation

- Source program translated ("compiled") to another language
  ⇒ generate executable machine code



Compilation

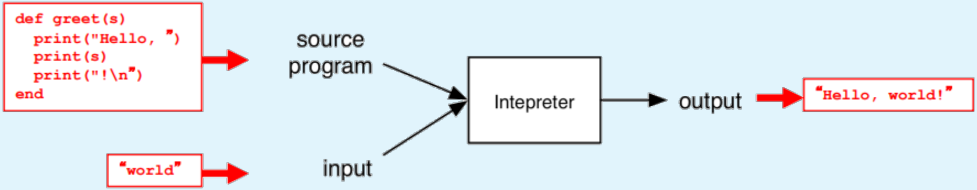
- Source program translated (“compiled”) to another language
  ⇒ generate executable machine code



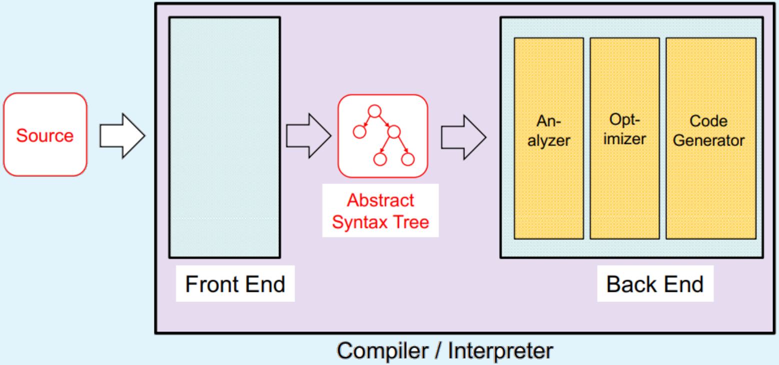
- Single translation but multiple executions
  ⇒ large amount of time code analysis and optimization
  ⇒ typically runs fast but hard to debug

Interpretation

- Interpreter executes each instruction in source program one step at a time
  $\Rightarrow$ no separate executable



Architecture (Compilers and Interpreters)



Font-End and Back-End

- Front ends handle syntax

  ⇒ Parser converts source code into intermediate format (“parse tree”) reflecting program structure

  ⇒ Static analyzer checks parse tree for errors (e.g., erroneous use of types), may also modify it
- Back ends handle semantics

  ⇒ Compiler back end (“code generator”): translates intermediate representation into “object language”

  ⇒ Interpreter back end: executes intermediate representation directly w.r.t. predefined semantics

Example

- gcc

  ⇒ Compiler – C code translated to object code, executed directly on hardware (as a separate step)
- sh/csh/tcsh/bash

  ⇒ Interpreter – commands executed by shell program

Compilers vs Interpreters

- Compilers

  ⇒ Generated code more efficient

  ⇒ “Heavy”

- Interpreters

  ⇒ Great for debugging, program property analyses

- In practice

  ⇒ “General-purpose” programming languages (e.g. C, Java) are often compiled; but debuggers provide interpreter support

  ⇒ Scripting languages and other special-purpose languages are interpreted

Attributes of a Good Language

- Portability of programs

  ⇒ Develop on one computer system, run on another

Attributes of a Good Language

- Portability of programs

  ⇒ Develop on one computer system, run on another
- Programming environment

  ⇒ External support for the language

  ⇒ Libraries, documentation, community, IDEs, ...

Attributes of a Good Language

- Portability of programs

  ⇒ Develop on one computer system, run on another
- Programming environment

  ⇒ External support for the language

  ⇒ Libraries, documentation, community, IDEs, ...
- Orthogonality

  ⇒ Every combination of features is meaningful

  ⇒ Features work independently

Attributes of a Good Language (cont'd)

- Support for abstraction

  ⇒ Hide details where you do not need them

  ⇒ Program data reflects the problem you are solving

Attributes of a Good Language (cont'd)

- Support for abstraction

  ⇒ Hide details where you do not need them

  ⇒ Program data reflects the problem you are solving
- Security and safety

  ⇒ Should be very difficult to write unsafe programs

Attributes of a Good Language (cont'd)

- Support for abstraction

  ⇒ Hide details where you do not need them

  ⇒ Program data reflects the problem you are solving
- Security and safety

  ⇒ Should be very difficult to write unsafe programs
- Ease of program verification

  ⇒ Does a program correctly perform its required function?

Thanks! & Questions?