

# General

- ▶ All information on the Web page  
`http://www.voronkov.com/lics.cgi`
- ▶ Assessment: exam (80%), exercises (20%).
- ▶ Exercises: at the end of (almost) every week with the deadline in one week.

# Computer Systems and Correctness

Suppose we design a (complex) system, which may contain various components, for example, sensors, networks, computers; all running software.

We have high requirements on the **correctness** of the system (**safety, reliability, security, availability, no deadlocks** etc.)

How can one ensure that the system satisfies these requirements?

Computer systems are becoming increasingly unreliable.

# Computer Systems and Correctness

Suppose we design a (complex) system, which may contain various components, for example, sensors, networks, computers; all running software.

We have high requirements on the **correctness** of the system (**safety, reliability, security, availability, no deadlocks** etc.)

How can one ensure that the system satisfies these requirements?

Computer systems are becoming increasingly unreliable.

# Computer Systems and Correctness

Suppose we design a (complex) system, which may contain various components, for example, sensors, networks, computers; all running software.

We have high requirements on the **correctness** of the system (**safety, reliability, security, availability, no deadlocks** etc.)

How can one ensure that **the system satisfies these requirements?**

Computer systems are becoming increasingly unreliable.

# Computer Systems and Correctness

Suppose we design a (complex) system, which may contain various components, for example, sensors, networks, computers; all running software.

We have high requirements on the **correctness** of the system (**safety, reliability, security, availability, no deadlocks** etc.)

How can one ensure that **the system satisfies these requirements?**

**Computer systems are becoming increasingly unreliable.**

## Small Example: Software

Consider the following fragment of a C program:

```
/* Returns a new array of integers of a given
   length initialised by a non-zero value */
int* allocateArray(int length)
{
    int i;
    int* array;
    array = malloc(sizeof(int)*length);

    for (i = 0; i <= length; i++)
        array[i] = 0;
    return array;
}
```

Is this program correct?

We discussed **correctness** of a program without ever defining what it means.

So what is correctness?

## Small Example: Software

Consider the following fragment of a C program:

```
/* Returns a new array of integers of a given
   length initialised by a non-zero value */
int* allocateArray(int length)
{
    int i;
    int* array;
    array = malloc(sizeof(int)*length);

    for (i = 0; i <= length; i++)
        array[i] = 0;
    return array;
}
```

Is this program correct?

**Hardly: it writes into memory that has not been allocated.**

We discussed **correctness** of a program without ever defining what it means.

So what is correctness?

## Small Example: Software

Consider the following fragment of a C program:

```
/* Returns a new array of integers of a given
   length initialised by a non-zero value */
int* allocateArray(int length)
{
    int i;
    int* array;
    array = malloc(sizeof(int)*length);

    for (i = 0; i < length; i++)
        array[i] = 0;
    return array;
}
```

Is this program correct?

We discussed **correctness** of a program without ever defining what it means.

So what is correctness?

## Small Example: Software

Consider the following fragment of a C program:

```
/* Returns a new array of integers of a given
   length initialised by a non-zero value */
int* allocateArray(int length)
{
    int i;
    int* array;
    array = malloc(sizeof(int)*length); // may return 0!

    for (i = 0; i < length; i++)
        array[i] = 0;
    return array;
}
```

Is this program correct?

**No: it may write to the null address.**

We discussed **correctness** of a program without ever defining what it means.

So what is correctness?

## Small Example: Software

Consider the following fragment of a C program:

```
/* Returns a new array of integers of a given
   length initialised by a non-zero value */
int* allocateArray(int length)
{
    int i;
    int* array;
    array = malloc(sizeof(int)*length);
    if (! array) return 0;
    for (i = 0; i < length; i++)
        array[i] = 0;
    return array;
}
```

Is this program correct?

We discussed **correctness** of a program without ever defining what it means.

So what is correctness?

## Small Example: Software

Consider the following fragment of a C program:

```
/* Returns a new array of integers of a given
   length initialised by a non-zero value */
int* allocateArray(int length)
{
    int i;
    int* array;
    array = malloc(sizeof(int)*length);
    if (! array) return 0;
    for (i = 0; i < length; i++)
        array[i] = 0;
    return array;
}
```

Is this program correct?

**No: it initialises the array by zeros**

We discussed **correctness** of a program without ever defining what it means.

So what is correctness?

## Small Example: Software

Consider the following fragment of a C program:

```
/* Returns a new array of integers of a given
   length initialised by a non-zero value */
int* allocateArray(int length)
{
    int i;
    int* array;
    array = malloc(sizeof(int)*length);
    if (! array) return 0;
    for (i = 0; i < length; i++)
        array[i] = 0;
    return array;
}
```

Is this program correct?

We discussed **correctness** of a program without ever defining what it means.

So what is correctness?

## Small Example: Software

Consider the following fragment of a C program:

```
/* Returns a new array of integers of a given
   length initialised by a non-zero value */
int* allocateArray(int length)
{
    int i;
    int* array;
    array = malloc(sizeof(int)*length);
    if (! array) return 0;
    for (i = 0; i < length; i++)
        array[i] = 0;
    return array;
}
```

Is this program correct?

We discussed **correctness** of a program without ever defining what it means.

**So what is correctness?**

# Notes

- ▶ We could spot the first two errors without knowing anything about the **intended meaning** of the program. But we had to understand the meaning of C programs in general and some specific properties of programming in C.
- ▶ To understand the last “error” we had to know something about the **the intended behaviour of the program**.

# Notes

- ▶ We could spot the first two errors without knowing anything about the **intended meaning** of the program. But we had to understand the meaning of C programs in general and some specific properties of programming in C.
- ▶ To understand the last “error” we had to know something about the **the intended behaviour of the program**.

## Another example: circuit design

We used a circuit  $C_1$  in a processor and would like to replace it by another circuit  $C_2$ . For example, we may believe that the use of  $C_2$  results in a lower energy consumption.

We want to be sure that  $C_2$  is correct, that is, it will behave according to some specification.

If we know that  $C_1$  is correct, it is sufficient to prove that  $C_2$  is functionally equivalent to  $C_1$ .

## Another example: circuit design

We used a circuit  $C_1$  in a processor and would like to replace it by another circuit  $C_2$ . For example, we may believe that the use of  $C_2$  results in a lower energy consumption.

We want **to be sure** that  $C_2$  is correct, that is, it will behave according to some specification.

If we know that  $C_1$  is correct, it is sufficient to **prove** that  $C_2$  is functionally equivalent to  $C_1$ .

## Another example: circuit design

We used a circuit  $C_1$  in a processor and would like to replace it by another circuit  $C_2$ . For example, we may believe that the use of  $C_2$  results in a lower energy consumption.

We want **to be sure** that  $C_2$  is correct, that is, it will behave according to some specification.

If we know that  $C_1$  is correct, it is sufficient to **prove** that  $C_2$  is functionally equivalent to  $C_1$ .

## Another example: Vending Machine

1. The vending machine contains a drink storage, a coin slot, and a drink dispenser. The drink storage stores drinks of two kinds: beer and coffee. We are only interested in whether a particular kind of drink is currently being stored or not, but not interested in the amount of it.
2. The coin slot can accommodate up to three coins.
3. The drink dispenser can store at most one drink. If it contains a drink, this drink should be removed before the next one can be dispensed.
4. A can of beer costs two coins. A cup of coffee costs one coin.
5. There are two kinds of customers: students and professors. Students drink only beer, professors drink only coffee.
6. From time to time the drink storage can be recharged.

Suppose that we would like to **prove** some properties of this model, for example that a student will never leave money in the coin slot.

## Another example: Vending Machine

1. The vending machine contains a drink storage, a coin slot, and a drink dispenser. The drink storage stores drinks of two kinds: beer and coffee. We are only interested in whether a particular kind of drink is currently being stored or not, but not interested in the amount of it.
2. The coin slot can accommodate up to three coins.
3. The drink dispenser can store at most one drink. If it contains a drink, this drink should be removed before the next one can be dispensed.
4. A can of beer costs two coins. A cup of coffee costs one coin.
5. There are two kinds of customers: students and professors. Students drink only beer, professors drink only coffee.
6. From time to time the drink storage can be recharged.

Suppose that we would like to **prove** some properties of this model, for example that a student will never leave money in the coin slot.

# How to establish correctness

- ▶ Build a **formal model** of the system;
- ▶ Find a **language** for expressing intended properties;
- ▶ The language must have a **semantics** that explains which models satisfy properties expressible in the language.
- ▶ Write a **specification**, that is, intended properties of the system in this language.
- ▶ **Prove** formally that the system model is also a model of the specification.

# How to establish correctness

- ▶ Build a **formal model** of the system;
- ▶ Find a **language** for expressing intended properties;
- ▶ The language must have a **semantics** that explains which models satisfy properties expressible in the language.
- ▶ Write a **specification**, that is, intended properties of the system in this language.
- ▶ **Prove** formally that the system model is also a model of the specification.

# How to establish correctness

- ▶ Build a **formal model** of the system;
- ▶ Find a **language** for expressing intended properties;
- ▶ The language must have a **semantics** that explains which models satisfy properties expressible in the language.
- ▶ Write a **specification**, that is, intended properties of the system in this language.
- ▶ **Prove** formally that the system model is also a model of the specification.

# How to establish correctness

- ▶ Build a **formal model** of the system;
- ▶ Find a **language** for expressing intended properties;
- ▶ The language must have a **semantics** that explains which models satisfy properties expressible in the language.
- ▶ Write a **specification**, that is, intended properties of the system in this language.
- ▶ **Prove** formally that the system model is also a model of the specification.

# How to establish correctness

- ▶ Build a **formal model** of the system;
- ▶ Find a **language** for expressing intended properties;
- ▶ The language must have a **semantics** that explains which models satisfy properties expressible in the language.
- ▶ Write a **specification**, that is, intended properties of the system in this language.
- ▶ **Prove** formally that the system model is also a model of the specification.

# What is logic?

- ▶ Syntax and semantics;
- ▶ Proof theory and model theory;
- ▶ Reasoning.

# Logic in computer science

- ▶ knowledge representation and reasoning;
- ▶ semantic Web;
- ▶ circuit design;
- ▶ constraint satisfaction;
- ▶ planning;
- ▶ software and hardware verification;
- ▶ databases (semantics and query optimisation);
- ▶ theorem proving in mathematics.

# This course

- ▶ propositional logic;
- ▶ satisfiability checking in propositional logic;
- ▶ semantic tableaux;
- ▶ binary decision diagrams (BDDs);
- ▶ quantified boolean formulas;
- ▶ propositional logic of finite domains;
- ▶ state-changing systems and transition systems;
- ▶ temporal logic;
- ▶ model checking.

# Propositional logic: syntax

Assume a countable set of **boolean variables**.

**Propositional formula:**

- ▶ Every boolean variable is a formula, also called **atomic formula**, or simply **atom**.
- ▶  $\top$  and  $\perp$  are formulas.
- ▶ If  $A_1, \dots, A_n$  are formulas, where  $n \geq 2$ , then  $(A_1 \wedge \dots \wedge A_n)$  and  $(A_1 \vee \dots \vee A_n)$  are formulas.
- ▶ If  $A$  is a formula, then  $\neg A$  is a formula.
- ▶ If  $A$  and  $B$  are formulas, then  $(A \rightarrow B)$  and  $(A \leftrightarrow B)$  are formulas.

# Propositional logic: syntax

Assume a countable set of **boolean variables**.

**Propositional formula:**

- ▶ Every boolean variable is a formula, also called **atomic formula**, or simply **atom**.
- ▶  $\top$  and  $\perp$  are formulas.
- ▶ If  $A_1, \dots, A_n$  are formulas, where  $n \geq 2$ , then  $(A_1 \wedge \dots \wedge A_n)$  and  $(A_1 \vee \dots \vee A_n)$  are formulas.
- ▶ If  $A$  is a formula, then  $\neg A$  is a formula.
- ▶ If  $A$  and  $B$  are formulas, then  $(A \rightarrow B)$  and  $(A \leftrightarrow B)$  are formulas.

The symbols  $\top, \perp, \wedge, \vee, \neg, \rightarrow, \leftrightarrow$  are called **connectives**.

# Propositional logic: syntax

Assume a countable set of **boolean variables**.

**Propositional formula:**

- ▶ Every boolean variable is a formula, also called **atomic formula**, or simply **atom**.
- ▶  $\top$  and  $\perp$  are formulas.
- ▶ If  $A_1, \dots, A_n$  are formulas, where  $n \geq 2$ , then  $(A_1 \wedge \dots \wedge A_n)$  and  $(A_1 \vee \dots \vee A_n)$  are formulas.
- ▶ If  $A$  is a formula, then  $\neg A$  is a formula.
- ▶ If  $A$  and  $B$  are formulas, then  $(A \rightarrow B)$  and  $(A \leftrightarrow B)$  are formulas.

The symbols  $\top, \perp, \wedge, \vee, \neg, \rightarrow, \leftrightarrow$  are called **connectives**.

# Propositional logic: syntax

Assume a countable set of **boolean variables**.

**Propositional formula:**

- ▶ Every boolean variable is a formula, also called **atomic formula**, or simply **atom**.
- ▶  $\top$  and  $\perp$  are formulas.
- ▶ If  $A_1, \dots, A_n$  are formulas, where  $n \geq 2$ , then  $(A_1 \wedge \dots \wedge A_n)$  and  $(A_1 \vee \dots \vee A_n)$  are formulas.
- ▶ If  $A$  is a formula, then  $\neg A$  is a formula.
- ▶ If  $A$  and  $B$  are formulas, then  $(A \rightarrow B)$  and  $(A \leftrightarrow B)$  are formulas.

The symbols  $\top, \perp, \wedge, \vee, \neg, \rightarrow, \leftrightarrow$  are called **connectives**.

# Connectives

Connective	Name	Priority
$\top$	verum	
$\perp$	falsum	
$\neg$	negation	4
$\wedge$	conjunction	3
$\vee$	disjunction	3
$\rightarrow$	implication	2
$\leftrightarrow$	equivalence	1

# Parsing Formulas

As usual, we will sometimes omit parenthesis in formulas and use priorities to disambiguate them. For example, in arithmetic we know that the expression

$$x \cdot y + 2 \cdot z$$

is equivalent to

$$(x \cdot y) + (2 \cdot z),$$

since  $\cdot$  has a **higher priority** than  $+$ .

Let us parse

$$\neg A \wedge B \rightarrow C \vee D \leftrightarrow E.$$

Connective	Priority
$\top$	
$\perp$	
$\neg$	4
$\wedge$	3
$\vee$	3
$\rightarrow$	2
$\leftrightarrow$	1

# Parsing Formulas

As usual, we will sometimes omit parenthesis in formulas and use priorities to disambiguate them. For example, in arithmetic we know that the expression

$$x \cdot y + 2 \cdot z$$

is equivalent to

$$(x \cdot y) + (2 \cdot z),$$

since  $\cdot$  has a **higher priority** than  $+$ .

We will also use priorities to disambiguate formulas as given in the following table. Let us parse

Connective	Priority
$\top$	
$\perp$	
$\neg$	4
$\wedge$	3
$\vee$	3
$\rightarrow$	2
$\leftrightarrow$	1

# Parsing Formulas

As usual, we will sometimes omit parenthesis in formulas and use priorities to disambiguate them. Let us parse

$$\neg A \wedge B \rightarrow C \vee D \leftrightarrow E.$$

Connective	Priority
$\top$	
$\perp$	
$\neg$	4
$\wedge$	3
$\vee$	3
$\rightarrow$	2
$\leftrightarrow$	1

Inside-out (starting with the highest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

Outside-in (starting with the lowest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

# Parsing Formulas

As usual, we will sometimes omit parenthesis in formulas and use priorities to disambiguate them. Let us parse

$$\neg A \wedge B \rightarrow C \vee D \leftrightarrow E.$$

Connective	Priority
$\top$	
$\perp$	
$\neg$	4
$\wedge$	3
$\vee$	3
$\rightarrow$	2
$\leftrightarrow$	1

Inside-out (starting with the highest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

Outside-in (starting with the lowest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

# Parsing Formulas

As usual, we will sometimes omit parenthesis in formulas and use priorities to disambiguate them. Let us parse

$$\neg A \wedge B \rightarrow C \vee D \leftrightarrow E.$$

Connective	Priority
$\top$	
$\perp$	
$\neg$	4
$\wedge$	3
$\vee$	3
$\rightarrow$	2
$\leftrightarrow$	1

Inside-out (starting with the highest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

Outside-in (starting with the lowest priority connectives):

$$(((\neg A \wedge B) \rightarrow (C \vee D)) \leftrightarrow E).$$

# Parsing Formulas

As usual, we will sometimes omit parenthesis in formulas and use priorities to disambiguate them. Let us parse

$$\neg A \wedge B \rightarrow C \vee D \leftrightarrow E.$$

Connective	Priority
$\top$	
$\perp$	
$\neg$	4
$\wedge$	3
$\vee$	3
$\rightarrow$	2
$\leftrightarrow$	1

Inside-out (starting with the highest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

Outside-in (starting with the lowest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

# Parsing Formulas

As usual, we will sometimes omit parenthesis in formulas and use priorities to disambiguate them. Let us parse

$$\neg A \wedge B \rightarrow C \vee D \leftrightarrow E.$$

Connective	Priority
$\top$	
$\perp$	
$\neg$	4
$\wedge$	3
$\vee$	3
$\rightarrow$	2
$\leftrightarrow$	1

Inside-out (starting with the highest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

Outside-in (starting with the lowest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

# Parsing Formulas

As usual, we will sometimes omit parenthesis in formulas and use priorities to disambiguate them. Let us parse

$$\neg A \wedge B \rightarrow C \vee D \leftrightarrow E.$$

Connective	Priority
$\top$	
$\perp$	
$\neg$	4
$\wedge$	3
$\vee$	3
$\rightarrow$	2
$\leftrightarrow$	1

Inside-out (starting with the highest priority connectives):

$$(((\neg A \wedge B) \rightarrow (C \vee D)) \leftrightarrow E).$$

Outside-in (starting with the lowest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

# Parsing Formulas

As usual, we will sometimes omit parenthesis in formulas and use priorities to disambiguate them. Let us parse

$$\neg A \wedge B \rightarrow C \vee D \leftrightarrow E.$$

Connective	Priority
$\top$	
$\perp$	
$\neg$	4
$\wedge$	3
$\vee$	3
$\rightarrow$	2
$\leftrightarrow$	1

Inside-out (starting with the highest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

Outside-in (starting with the lowest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

# Parsing Formulas

As usual, we will sometimes omit parenthesis in formulas and use priorities to disambiguate them. Let us parse

$$\neg A \wedge B \rightarrow C \vee D \leftrightarrow E.$$

Connective	Priority
$\top$	
$\perp$	
$\neg$	4
$\wedge$	3
$\vee$	3
$\rightarrow$	2
$\leftrightarrow$	1

Inside-out (starting with the highest priority connectives):

$$((\neg A \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

Outside-in (starting with the lowest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

# Parsing Formulas

As usual, we will sometimes omit parenthesis in formulas and use priorities to disambiguate them. Let us parse

$$\neg A \wedge B \rightarrow C \vee D \leftrightarrow E.$$

Connective	Priority
$\top$	
$\perp$	
$\neg$	4
$\wedge$	3
$\vee$	3
$\rightarrow$	2
$\leftrightarrow$	1

Inside-out (starting with the highest priority connectives):

$$((\neg A \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

Outside-in (starting with the lowest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

# Parsing Formulas

As usual, we will sometimes omit parenthesis in formulas and use priorities to disambiguate them. Let us parse

$$\neg A \wedge B \rightarrow C \vee D \leftrightarrow E.$$

Connective	Priority
$\top$	
$\perp$	
$\neg$	4
$\wedge$	3
$\vee$	3
$\rightarrow$	2
$\leftrightarrow$	1

Inside-out (starting with the highest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

Outside-in (starting with the lowest priority connectives):

$$(((\neg A) \wedge B) \rightarrow (C \vee D)) \leftrightarrow E.$$

# Semantics, Interpretation

Consider an arithmetical expression, for example

$$x \cdot y + 2 \cdot z.$$

In arithmetic the meaning of expressions with variables is defined as follows.

Take a mapping from variables (integer) values, for example

$$\{x \mapsto 1, y \mapsto 7, z \mapsto -3\}.$$

Then, under this mapping the expression has the value 1. In other words, when we interpret variables as values, we can compute the value of the expression.

# Semantics, Interpretation

Consider an arithmetical expression, for example

$$x \cdot y + 2 \cdot z.$$

In arithmetic the meaning of expressions with variables is defined as follows.

Take a mapping from variables (integer) values, for example

$$\{x \mapsto 1, y \mapsto 7, z \mapsto -3\}.$$

Then, under this mapping the expression has the value 1. In other words, when we interpret variables as values, we can compute the value of the expression.

# Semantics, Interpretation

Consider an arithmetical expression, for example

$$x \cdot y + 2 \cdot z.$$

In arithmetic the meaning of expressions with variables is defined as follows.

Take a mapping from variables (integer) values, for example

$$\{x \mapsto 1, y \mapsto 7, z \mapsto -3\}.$$

Then, under this mapping the expression has the value **1**. In other words, when we interpret variables as values, we can compute the value of the expression.

# Semantics, Interpretation

Likewise, the semantics of propositional formulas can be defined by assigning boolean values to variables.

- ▶ A **boolean value**, also called a **truth value**, is either **true** (denoted 1) or **false** (denoted 0).
- ▶ An **interpretation** for a set  $P$  of boolean variables is a mapping  $I : P \rightarrow \{1, 0\}$ .
- ▶ Interpretations are also called **truth assignments**.

# Semantics, Interpretation

Likewise, the semantics of propositional formulas can be defined by assigning boolean values to variables.

- ▶ A **boolean value**, also called a **truth value**, is either **true** (denoted **1**) or **false** (denoted **0**).
- ▶ An **interpretation** for a set  $P$  of boolean variables is a mapping  $I: P \rightarrow \{1, 0\}$ .
- ▶ Interpretations are also called **truth assignments**.

# Semantics, Interpretation

Likewise, the semantics of propositional formulas can be defined by assigning boolean values to variables.

- ▶ A **boolean value**, also called a **truth value**, is either **true** (denoted 1) or **false** (denoted 0).
- ▶ An **interpretation** for a set  $P$  of boolean variables is a mapping  $I : P \rightarrow \{1, 0\}$ .
- ▶ Interpretations are also called **truth assignments**.

# Semantics, Interpretation

Likewise, the semantics of propositional formulas can be defined by assigning boolean values to variables.

- ▶ A **boolean value**, also called a **truth value**, is either **true** (denoted 1) or **false** (denoted 0).
- ▶ An **interpretation** for a set  $P$  of boolean variables is a mapping  $I : P \rightarrow \{1, 0\}$ .
- ▶ Interpretations are also called **truth assignments**.