Software Correctness: The Construction of Correct Software

Introduction

Stefan Hallerstede (sha@ece.au.dk)
Carl Peter Leslie Schultz (cschultz@ece.au.dk)

John Hatcliff (Kansas State University)
Robby (Kansas State University)

Organisation

Course Topic and Scope
- Motivation
- Slang and Sireum/Logika
- Installing Sireum/Logika

Reasoning about Programs
- Example A: Observing a Logical Error
- Example B: Observing Undefined Behaviour

Methodology
- Formality
- Contracts and Reasoning Support
- A First Look at Contracts

The Programming Project

Summary

Organisation

Course Topic and Scope
- Motivation
- Slang and Sireum/Logika
- Installing Sireum/Logika

Reasoning about Programs
- Example A: Observing a Logical Error
- Example B: Observing Undefined Behaviour

Methodology
- Formality
- Contracts and Reasoning Support
- A First Look at Contracts

The Programming Project

Summary

General Information

- Lecturers
  - Stefan Hallerstede (sha@ece.au.dk)
  - Hugo Daniel Macedo (hdm@ece.au.dk)
- Time and place
  - Tuesday, 8:00 – 12:00
  - Room 423
  - Edison (5125) building
- In-class exercises
- Multi-paradigm programming project
  - In groups
  - At home
- Exam
  - Oral
  - No aids
  - 15 minutes + 5 minutes deliberation

Schedule

1. Introduction
2. Tracing Facts
3. Conditionals
4. Contracts (Test)
5. Contracts (Proof)
6. — *Project (Q/A)*
7. Loops and Recursion
8. Loop Unfolding
9. Loop Testing
10. — *Project (Q/A)*
11. Sequences and Quantification
12. Immutable Structures
13. Mutable Structures
14. Programs as Proofs — *Project Demo*

Programming Project

- Solve an interesting problem and program the solution
- Java, Scala and Slang (more on Slang later)
  - Java: `https://dev.java/learn/`
  - Scala: `https://docs.scala-lang.org`
- For the project you can use any IDE of your choice
- In the lectures we use Sireum, which is based on IntelliJ IDEA
  - IDEA: `https://www.jetbrains.com/idea/`
  - Sireum: `https://sireum.org`
- In the project you
  - practice methods and techniques discussed in the lecture
  - train your programming skills
  - enjoy the company of your fellow students
  - have fun!

Organisation

Course Topic and Scope
- Motivation
- Slang and Sireum/Logika
- Installing Sireum/Logika

Reasoning about Programs
- Example A: Observing a Logical Error
- Example B: Observing Undefined Behaviour

Methodology
- Formality
- Contracts and Reasoning Support
- A First Look at Contracts

The Programming Project

Summary

Motivation

- View programming as an **engineering** activity to **construct correct software**
- Engineers ensure that
  - they are certain that an artifact works **before** it is delivered
  - they understand **why** it works
  - **others** can also understand why it works
  - others can **maintain** the artefact based on this knowledge
- There are **established ways** that engineers ensure the above. They
  - produce a **specification** of the artefact required to be built
  - put forward an **argument** that shows how the artefact satisfies the specification
    (The argument aids understanding and explains why the artefact works.)
  - include **documentation** that makes the argument accessible to others
- The artefacts software engineers build are **software**, of course

Specification, Argumentation and Documentation

- Distinguish
  - **Specification** stating **what** functionality is required
  - **Implementation** stating **how** that functionality is achieved
  - In this course we are concerned with **both**, specification and implementation
- Argumentation ensures that how functionality is achieved is what is required
- An argument must **relate** specification and implementation in such way, that
  - the implementation **achieves** all the functionality required by the specification or, said differently,
  - the functionality of the implementation **does not deviate** from what is specified
- We say, the implementation must be **correct** with respect to the specification

- Aside: you already know how to detect deviations from specified behaviour: **testing.**

  That all functionality is achieved can be verified by **proof.**

  In short, testing looks for the **presence** of defects and proof shows their **absence.**

From Scala to Slang

- **Scala** is a modern feature-rich programming language.
- It offers sufficient support concerning our interest in **programming**.
- It **lacks** support for dealing with correctness.
- This is achieved by means of the Scala dialect **Slang**
- In a nutshell, Slang is a subset of Scala with that support added
  (Of course, this is a gross oversimplification! – but good to remember.)
- Syntactically, from out viewpoint Scala and Slang are **nearly identical**.
- So, it’s easy to **switch between the two**
  and apply what you’ve learned in one to the other one

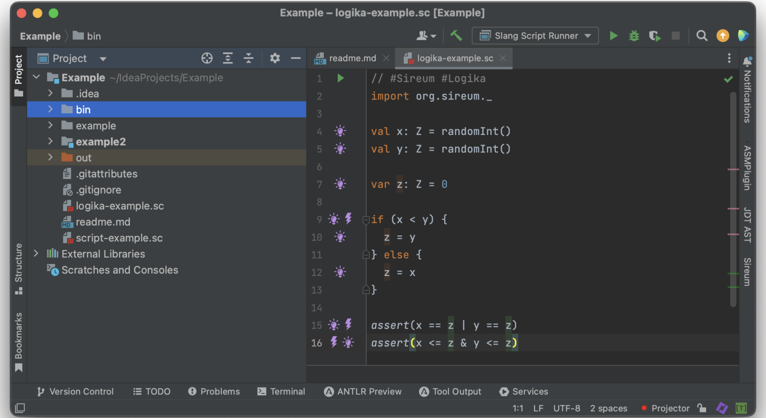
Sireum/Logika

- **Sireum** is an integrated verification environment (IVE).
- It contains a component called **Logika**
  that supports **automated reasoning** about Slang programs.
- It is based on IntelliJ IDEA.
- So, one can also use it for **programming** in Java and Scala
  with all the customary programming support.

- **Automation** is necessary to master more complex and changing programs
- **Resemblance** to a common IDE is helpful to avoid extra learning effort
- A conference presentation video on Sireum/Logika:
  Integrated Formal Verification Environment for seL4 Applications
  Robby, J. Hatcliff, T. Carpenter, D. Stewart

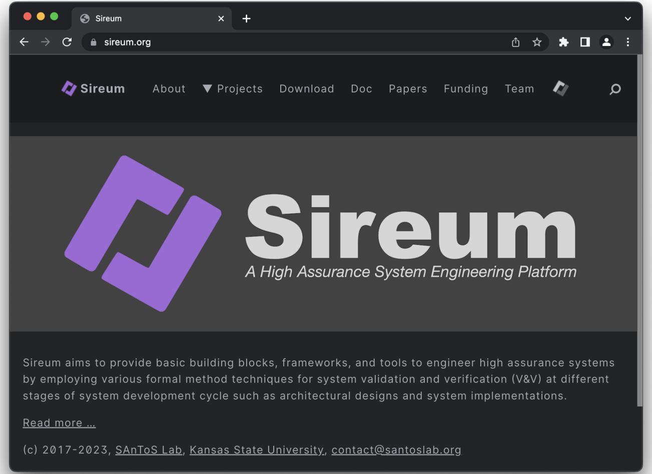
The Sireum/Logika IVE

- Screenshot of the Sireum/Logika IVE
- On the right-hand side a verified Slang program is shown



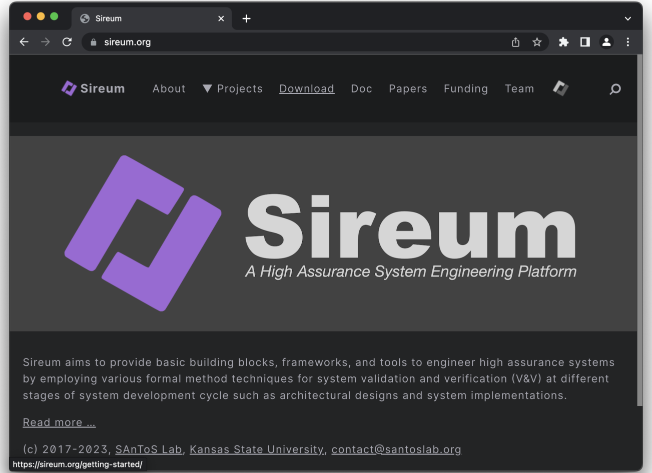
Installing Sireum/Logika

- Go to the web site `https://sireum.org`



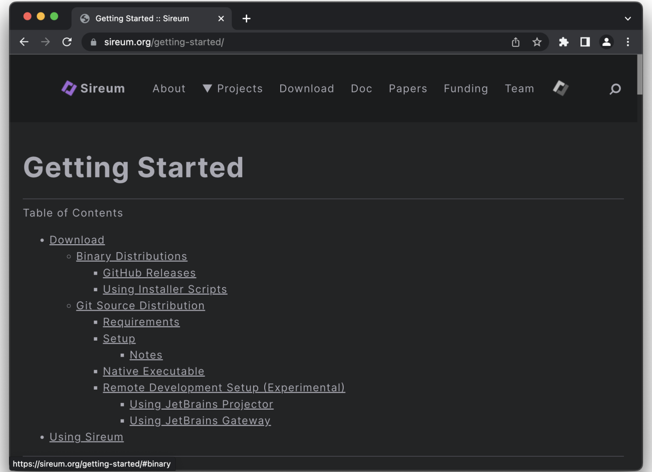
Installing Sireum/Logika

- Choose “Download”



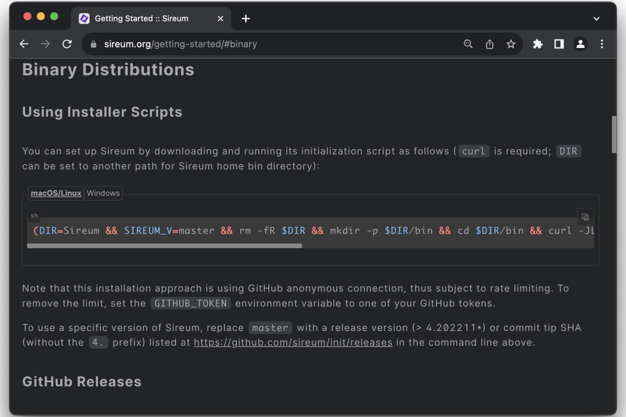
Installing Sireum/Logika

- Choose “Binary Distributions”



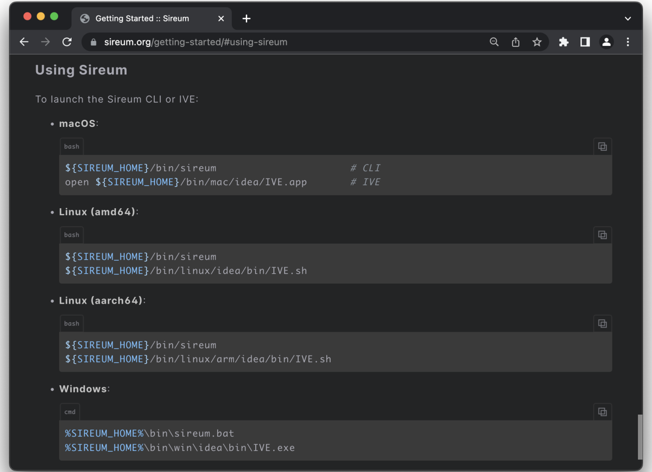
Installing Sireum/Logika

- Use the installer script



Installing Sireum/Logika

- Try it out



Using Sireum

To launch the Sireum CLI or IVE:

- **macOS**:

```bash
${SIREUM_HOME}/bin/sireum                      # CLI
open ${SIREUM_HOME}/bin/mac/idea/IVE.app       # IVE
```

- **Linux (amd64)**:

```bash
${SIREUM_HOME}/bin/sireum
${SIREUM_HOME}/bin/linux/idea/bin/IVE.sh
```

- **Linux (aarch64)**:

```bash
${SIREUM_HOME}/bin/sireum
${SIREUM_HOME}/bin/linux/arm/idea/bin/IVE.sh
```

- **Windows**:

```cmd
%SIREUM_HOME%\bin\sireum.bat
%SIREUM_HOME%\bin\win\idea\bin\IVE.exe
```

Organisation

Course Topic and Scope
- Motivation
- Slang and Sireum/Logika
- Installing Sireum/Logika

Reasoning about Programs
- Example A: Observing a Logical Error
- Example B: Observing Undefined Behaviour

Methodology
- Formality
- Contracts and Reasoning Support
- A First Look at Contracts

The Programming Project

Summary

Reasoning about Programs: Example A

- Suppose, the following program is expected to yield `z == 3`

```
val x: Z = 1
val y: Z = x + x + 1
val z: Z = x + y
```

Reasoning about Programs: Example A

- We can assert `z == 3` at the end of the program to ensure this

```
val x: Z = 1
val y: Z = x + x + 1
val z: Z = x + y
assert(z == 3)
```

Reasoning about Programs: Example A

- We can assert `z == 3` at the end of the program to ensure this

```
val x: Z = 1
val y: Z = x + x + 1
val z: Z = x + y
assert(z == 3)
```

- The program aborts if `z != 3` and terminates normally otherwise

Reasoning about Programs: Example A

- We can assert `z == 3` at the end of the program to ensure this

```
val x: Z = 1
val y: Z = x + x + 1
val z: Z = x + y
assert(z == 3)
```

- The program aborts if `z != 3` and terminates normally otherwise
- Does the program abort or terminate normally?

Reasoning about Programs: Example A

- We can assert `z == 3` at the end of the program to ensure this

```
val x: Z = 1
val y: Z = x + x + 1
val z: Z = x + y
assert(z == 3)
```

- The program aborts if `z != 3` and terminates normally otherwise
- Does the program abort or terminate normally?
- This is surprisingly non-obvious!

Reasoning about Programs: Example A

- We can assert `z == 3` at the end of the program to ensure this

```
val x: Z = 1
val y: Z = x + x + 1
val z: Z = x + y
assert(z == 3)
```

- The program aborts if `z != 3` and terminates normally otherwise
- Does the program abort or terminate normally?
- This is surprisingly non-obvious!
- Similarly, trivial but non-obvious off-by-one errors are very common in programming (Discussion: Why are off-by-one errors so common and what can we do to prevent it?)

Reasoning about Programs: Example A

- Let's reason about the program

```
val x: Z = 1
val y: Z = x + x + 1
val z: Z = x + y
assert(z == 3)
```

Reasoning about Programs: Example A

- Let's reason about the program

```
val x: Z = 1
val y: Z = x + x + 1
val z: Z = x + y
assert(z == 3)
```

- We can do this by observing the values of the variables following the assignments

Reasoning about Programs: Example A

- Let's reason about the program

```
val x: Z = 1
val y: Z = x + x + 1
val z: Z = x + y
assert(z == 3)
```

- We can do this by observing the values of the variables following the assignments
- After the first assignment `x` has the value `1`

Reasoning about Programs: Example A

- Let's reason about the program

```
val x: Z = 1
val y: Z = x + x + 1
val z: Z = x + y
assert(z == 3)
```

- We can do this by observing the values of the variables following the assignments
- After the first assignment `x` has the value `1`
- Let's record this by adding an assertion

Reasoning about Programs: Example A

- Based on this insight we continue reasoning

```
val x: Z = 1
assert(x == 1)
val y: Z = x + x + 1
val z: Z = x + y
assert(z == 3)
```

Reasoning about Programs: Example A

- Based on this insight we continue reasoning

```
val x: Z = 1
assert(x == 1)
val y: Z = x + x + 1
val z: Z = x + y
assert(z == 3)
```

- Now, we know the value of `y` after seconds assignment: `3`

Reasoning about Programs: Example A

- Based on this insight we continue reasoning

```
val x: Z = 1
assert(x == 1)
val y: Z = x + x + 1
val z: Z = x + y
assert(z == 3)
```

- Now, we know the value of `y` after seconds assignment: `3`
- We also know that the value of `x` remains unchanged

Reasoning about Programs: Example A

- Based on this insight we continue reasoning

```
val x: Z = 1
assert(x == 1)
val y: Z = x + x + 1
val z: Z = x + y
assert(z == 3)
```

- Now, we know the value of `y` after seconds assignment: `3`
- We also know that the value of `x` remains unchanged
- Let's record this by adding two assertions

Reasoning about Programs: Example A

- Now, we're in position to determine the value of `z`

```
val x: Z = 1
assert(x == 1)
val y: Z = x + x + 1
assert(x == 1)
assert(y == 3)
val z: Z = x + y
assert(z == 3)
```

Reasoning about Programs: Example A

- Now, we're in position to determine the value of `z`

```
val x: Z = 1
assert(x == 1)
val y: Z = x + x + 1
assert(x == 1)
assert(y == 3)
val z: Z = x + y
assert(z == 3)
```

- Since, `x` equals `1` and `y` equals `3`, we infer that `z` must equal `4` after the last assignment

Reasoning about Programs: Example A

- Now, we're in position to determine the value of `z`

```
val x: Z = 1
assert(x == 1)
val y: Z = x + x + 1
assert(x == 1)
assert(y == 3)
val z: Z = x + y
assert(z == 3)
```

- Since, `x` equals `1` and `y` equals `3`, we infer that `z` must equal `4` after the last assignment
- Let's record this by adding one more assertion

Reasoning about Programs: Example A

- With the knowledge we have gained so far, we can say that the program aborts

```
val x: Z = 1
assert(x == 1)
val y: Z = x + x + 1
assert(x == 1)
assert(y == 3)
val z: Z = x + y
assert(z == 4)
assert(z == 3)
```

Reasoning about Programs: Example A

- With the knowledge we have gained so far, we can say that the program aborts

```
val x: Z = 1
assert(x == 1)
val y: Z = x + x + 1
assert(x == 1)
assert(y == 3)
val z: Z = x + y
assert(z == 4)
assert(z == 3)
```

- The first assertion about `z` succeeds: we know `z` equals `4`

Reasoning about Programs: Example A

- With the knowledge we have gained so far, we can say that the program aborts

```
val x: Z = 1
assert(x == 1)
val y: Z = x + x + 1
assert(x == 1)
assert(y == 3)
val z: Z = x + y
assert(z == 4)
assert(z == 3)
```

- The first assertion about `z` succeeds: we know `z` equals `4`
- As a consequence the second assertion about `z` cannot succeed and must abort

Reasoning about Programs: Example A

- With the knowledge we have gained so far, we can say that the program aborts

```
val x: Z = 1
assert(x == 1)
val y: Z = x + x + 1
assert(x == 1)
assert(y == 3)
val z: Z = x + y
assert(z == 4)
assert(z == 3)
```

- The first assertion about `z` succeeds: we know `z` equals `4`
- As a consequence the second assertion about `z` cannot succeed and must abort
- We know that `4 != 3`

Reasoning about Programs: Example A

- Had we considered the following program where `y` is assigned `x + 1`

```
val x: Z = 1
assert(x == 1)
val y: Z = x + 1
assert(x == 1)
assert(y == 2)
val z: Z = x + y
assert(z == 3)
```

Reasoning about Programs: Example A

- Had we considered the following program where `y` is assigned `x + 1`

```
val x: Z = 1
assert(x == 1)
val y: Z = x + 1
assert(x == 1)
assert(y == 2)
val z: Z = x + y
assert(z == 3)
```

  we would have confirmed that `z` equals `3` with the program terminating normally
- Note, that with the assertions inserted in the program it is easy to see this
- The assertions capture part of our argument and document it
- Reasoning can be local:
  - we only need to look at an assignment and the assertions just before and just after it
  - it is *compositional*

Reasoning about Programs: Example B

- Suppose, the following program is expected to yield `z == 2`

```
val x: Z = 2
val y: Z = 0
val z: Z = x / y
assert(z == 2)
```

Reasoning about Programs: Example B

- Suppose, the following program is expected to yield `z == 2`

```
val x: Z = 2
val y: Z = 0
val z: Z = x / y
assert(z == 2)
```

- Does the program terminate normally or abort?

Reasoning about Programs: Example B

- Suppose, the following program is expected to yield `z == 2`

```
val x: Z = 2
val y: Z = 0
val z: Z = x / y
assert(z == 2)
```

- Does the program terminate normally or abort?
- As before, we reason about variables of the program and insert assertions to document this
- After the first assignment `x` equals `2`

Reasoning about Programs: Example B

- Having determined the new fact about `x` we continue with the second assignment

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
val z: Z = x / y
assert(z == 2)
```

Reasoning about Programs: Example B

- Having determined the new fact about `x` we continue with the second assignment

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
val z: Z = x / y
assert(z == 2)
```

- As before, we reason about variables of the program and insert assertions to document this
- After the first assignment `x` equals `2`

Reasoning about Programs: Example B

- Let's continue with the assignment to `y`

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
val z: Z = x / y
assert(z == 2)
```

Reasoning about Programs: Example B

- Let's continue with the assignment to `y`

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
val z: Z = x / y
assert(z == 2)
```

- After the second assignment `y` equals `0`
- The value of `x` remains unchanged

Reasoning about Programs: Example B

- Having collected all necessary facts, we can consider the third assignment

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
assert(x == 2)
assert(y == 0)
val z: Z = x / y
assert(z == 2)
```

Reasoning about Programs: Example B

- Having collected all necessary facts, we can consider the third assignment

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
assert(x == 2)
assert(y == 0)
val z: Z = x / y
assert(z == 2)
```

- In order to compute `x / y` it is *required* that `y` is different from `0`

Reasoning about Programs: Example B

- Having collected all necessary facts, we can consider the third assignment

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
assert(x == 2)
assert(y == 0)
val z: Z = x / y
assert(z == 2)
```

- In order to compute `x / y` it is *required* that `y` is different from `0`
- This condition must be *ensured* before `x / y` is evaluated

Reasoning about Programs: Example B

- Having collected all necessary facts, we can consider the third assignment

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
assert(x == 2)
assert(y == 0)
val z: Z = x / y
assert(z == 2)
```

- In order to compute `x / y` it is *required* that `y` is different from `0`
- This condition must be *ensured* before `x / y` is evaluated
- Let's add an assertion for the required condition

Reasoning about Programs: Example B

- With this knowledge we can determine that the program aborts

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
assert(x == 2)
assert(y == 0)
assert(y != 0)
val z: Z = x / y
assert(z == 2)
```

Reasoning about Programs: Example B

- With this knowledge we can determine that the program aborts

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
assert(x == 2)
assert(y == 0)
assert(y != 0)
val z: Z = x / y
assert(z == 2)
```

- The assertion `y == 0` succeeds and the assertion `y != 0` fails

Reasoning about Programs: Example B

- With this knowledge we can determine that the program aborts

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
assert(x == 2)
assert(y == 0)
assert(y != 0)
val z: Z = x / y
assert(z == 2)
```

- The assertion `y == 0` succeeds and the assertion `y != 0` fails
- Hence, the program aborts

Reasoning about Programs: Example B

- With this knowledge we can determine that the program aborts

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
assert(x == 2)
assert(y == 0)
assert(y != 0)
val z: Z = x / y
assert(z == 2)
```

Reasoning about Programs: Example B

- With this knowledge we can determine that the program aborts

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
assert(x == 2)
assert(y == 0)
assert(y != 0)
val z: Z = x / y
assert(z == 2)
```

- Just as in the preceding case we have discovered that an expected fact is contradicted by an observed fact

Reasoning about Programs: Example B

- With this knowledge we can determine that the program aborts

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
assert(x == 2)
assert(y == 0)
assert(y != 0)
val z: Z = x / y
assert(z == 2)
```

- Just as in the preceding case we have discovered that an expected fact is contradicted by an observed fact
- Such a contradiction shows to us that the program is not correct

Reasoning about Programs: Example B

- With this knowledge we can determine that the program aborts

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
assert(x == 2)
assert(y == 0)
assert(y != 0)
val z: Z = x / y
assert(z == 2)
```

- Just as in the preceding case we have discovered that an expected fact is contradicted by an observed fact
- Such a contradiction shows to us that the program is not correct
- It does not yield the expected result

Reasoning about Programs: Example B

- With this knowledge we can determine that the program aborts

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
assert(x == 2)
assert(y == 0)
assert(y != 0)
val z: Z = x / y
assert(z == 2)
```

- Just as in the preceding case we have discovered that an expected fact is contradicted by an observed fact
- Such a contradiction shows to us that the program is not correct
- It does not yield the expected result
- The methods discussed in this course reveal such contradictions

Reasoning about Programs: Example B

- With this knowledge we can determine that the program aborts

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
assert(x == 2)
assert(y == 0)
assert(y != 0)
val z: Z = x / y
assert(z == 2)
```

- Just as in the preceding case we have discovered that an expected fact is contradicted by an observed fact
- Such a contradiction shows to us that the program is not correct
- It does not yield the expected result
- The methods discussed in this course reveal such contradictions
- We consider both testing and proof from a logical point of view

Reasoning about Programs: Example B

- With this knowledge we can determine that the program aborts

```
val x: Z = 2
assert(x == 2)
val y: Z = 0
assert(x == 2)
assert(y == 0)
assert(y != 0)
val z: Z = x / y
assert(z == 2)
```

- Just as in the preceding case we have discovered
  that an expected fact is contradicted by an observed fact
- Such a contradiction shows to us that the program is not correct
- It does not yield the expected result
- The methods discussed in this course reveal such contradictions
- We consider both testing and proof from a logical point of view
- We are interested in the reasoning behind them and in the knowledge gained

Reasoning about Programs: Example B

- Had we analysed the program below where `y` is assigned `1`

```
val x: Z = 2
assert(x == 2)
val y: Z = 1
assert(x == 2)
assert(y == 1)
assert(y != 0)
val z: Z = x / y
assert(z == 2)
```

Reasoning about Programs: Example B

- Had we analysed the program below where y is assigned 1

```
val x: Z = 2
assert(x == 2)
val y: Z = 1
assert(x == 2)
assert(y == 1)
assert(y != 0)
val z: Z = x / y
assert(z == 2)
```

the last assertion would have been confirmed to hold

and we would have proved that this program always terminates normally

Organisation

Course Topic and Scope
- Motivation
- Slang and Sireum/Logika
- Installing Sireum/Logika

Reasoning about Programs
- Example A: Observing a Logical Error
- Example B: Observing Undefined Behaviour

Methodology
- Formality
- Contracts and Reasoning Support
- A First Look at Contracts

The Programming Project

Summary

Formal and Informal Reasoning

- Our reasoning above was entirely informal.
- Even though we have stated some facts formally, such as

  ```
  z == 4
  ```

  we have argued informally.

Formal and Informal Reasoning

- Our reasoning above was entirely informal.
- Even though we have stated some facts formally, such as

```
z == 4
```

  we have argued informally.
- This is a good way to understand why a programs works or does not work.
- It helps us understand the program at hand.

Formal and Informal Reasoning

- Our reasoning above was entirely informal.
- Even though we have stated some facts formally, such as

```
z == 4
```

  we have argued informally.
- This is a good way to understand why a programs works or does not work.
- It helps us understand the program at hand.
- Although we have gained **some insights** from the approach above, it has **some flaws**
  - The use of `assert` **affects the execution** of the program
  - Its use was **not** very systematic, **neither** was the reasoning, so we're unsure what we have learned

Formal and Informal Reasoning

- Our reasoning above was entirely informal.
- Even though we have stated some facts formally, such as

```
z == 4
```

  we have argued informally.
- This is a good way to understand why a programs works or does not work.
- It helps us understand the program at hand.
- Although we have gained **some insights** from the approach above, it has **some flaws**
  - The use of `assert` **affects the execution** of the program
  - Its use was **not** very systematic, **neither** was the reasoning, so we're unsure what we have learned
  - For larger programs it would become difficult to master all the details
- Thus, we need **more rigour** and **more automation**
- This is not an argument against informality, but it has its limitations.

Programming at Larger Scales

- Consider programs with 100, 1000, . . . , 100000 or more lines of Slang code.

Programming at Larger Scales

- Consider programs with 100, 1000, . . . , 100000 or more lines of Slang code.
- The **larger** the **programs** get, the **more** we depend on **automation.**
- This is true, independently of whether we talk about testing or proof.

Programming at Larger Scales

- Consider programs with 100, 1000, . . . , 100000 or more lines of Slang code.
- The **larger** the **programs** get, the **more** we depend on **automation**.
- This is true, independently of whether we talk about testing or proof.
- **Automation** is only possible if the methods we apply are **rigorous and systematic**.

Programming at Larger Scales

- Consider programs with 100, 1000, . . . , 100000 or more lines of Slang code.
- The **larger** the **programs** get, the **more** we depend on **automation.**
- This is true, independently of whether we talk about testing or proof.
- **Automation** is only possible if the methods we apply are **rigorous and systematic**.
  - In this course we take ‘rigorous’ to mean ‘formal’.
  - Our take on ‘systematic’ is centred around the concepts of ‘contract‘ and direct reasoning support in Slang itself

Programming at Larger Scales

- Consider programs with 100, 1000, . . . , 100000 or more lines of Slang code.
- The **larger** the **programs** get, the **more** we depend on **automation**.
- This is true, independently of whether we talk about testing or proof.
- **Automation** is only possible if the methods we apply are **rigorous and systematic**.
  - In this course we take ‘rigorous’ to mean ‘formal’.
  - Our take on ‘systematic’ is centred around the concepts of ‘contract‘ and direct reasoning support in Slang itself
- This approach delivers the required automation and has additional benefits
  - Specification: contracts embed specifications in the program
  - Argumentation: Slang reasoning constructs make the argumentation explicit
  - Documentation: Contracts and reasoning support are included in Slang programs

Programming at Larger Scales

- Consider programs with 100, 1000, . . . , 100000 or more lines of Slang code.
- The **larger** the **programs** get, the **more** we depend on **automation**.
- This is true, independently of whether we talk about testing or proof.
- **Automation** is only possible if the methods we apply are **rigorous and systematic**.
  - In this course we take ‘rigorous’ to mean ‘formal’.
  - Our take on ‘systematic’ is centred around the concepts of ‘contract‘ and direct reasoning support in Slang itself
- This approach delivers the required automation and has additional benefits
  - Specification: contracts embed specifications in the program
  - Argumentation: Slang reasoning constructs make the argumentation explicit
  - Documentation: Contracts and reasoning support are included in Slang programs
- All three are kept in synchrony with Slang programs with the help of automated reasoning

Division Again

- We have seen a use of the division operation in Example B

Division Again

- We have seen a use of the division operation in Example B
- Using `assert` we can specify
  which facts are true before and after execution of the operation

Division Again

- We have seen a use of the division operation in Example B
- Using `assert` we can specify
  which facts are true before and after execution of the operation

```
assert(y != 0)
val z: Z = x / y
assert(z == x / y)
```

Division Again

- We have seen a use of the division operation in Example B
- Using `assert` we can specify
  which facts are true before and after execution of the operation

```
assert(y != 0)
val z: Z = x / y
assert(z == x / y)
```

- Before `z = x / y` is executed `y != 0` must be true
  and afterwards `z == x / y` is true

Division Again

- We have seen a use of the division operation in Example B
- Using `assert` we can specify
  which facts are true before and after execution of the operation

```
assert(y != 0)
val z: Z = x / y
assert(z == x / y)
```

- Before `z = x / y` is executed `y != 0` must be true
  and afterwards `z == x / y` is true
- If we consider `z = x / y` an operation
  developed separately from the context where it is used,
  we cannot use `assert` as the first statement, because we do not know anything about `y`

Division Again

- We have seen a use of the division operation in Example B
- Using `assert` we can specify
  which facts are true before and after execution of the operation

```
assert(y != 0)
val z: Z = x / y
assert(z == x / y)
```

- Before `z = x / y` is executed `y != 0` must be true
  and afterwards `z == x / y` is true
- If we consider `z = x / y` an operation
  developed separately from the context where it is used,
  we cannot use `assert` as the first statement, because we do not know anything about `y`
- All we can do then is to assume `y != 0`

Division Again

- We have seen a use of the division operation in Example B
- Using `assert` we can specify
  which facts are true before and after execution of the operation

```
assert(y != 0)
val z: Z = x / y
assert(z == x / y)
```

- Before `z = x / y` is executed `y != 0` must be true
  and afterwards `z == x / y` is true
- If we consider `z = x / y` an operation
  developed separately from the context where it is used,
  we cannot use `assert` as the first statement, because we do not know anything about `y`
- All we can do then is to assume `y != 0`
- For this purpose Slang contains the `assume` statement

Assume What You Require, Assert What You Ensure

- Using this we bracket the assignment `val z: Z = x / y` in a pair consisting of an `assume` and an `assert` statement

Assume What You Require, Assert What You Ensure

- Using this we bracket the assignment `val z: Z = x / y` in a pair consisting of an `assume` and an `assert` statement

```
assume(y != 0)
val z: Z = x / y
assert(z == x / y)
```

Assume What You Require, Assert What You Ensure

- Using this we bracket the assignment `val z: Z = x / y` in a pair consisting of an `assume` and an `assert` statement

```
assume(y != 0)
val z: Z = x / y
assert(z == x / y)
```

- Concerning its execution `assume` is not distinguishable from `assert` in Slang

Assume What You Require, Assert What You Ensure

- Using this we bracket the assignment `val z: Z = x / y` in a pair consisting of an `assume` and an `assert` statement

```
assume(y != 0)
val z: Z = x / y
assert(z == x / y)
```

- Concerning its execution `assume` is not distinguishable from `assert` in Slang
- However, concerning the reasoning they are very different:
  - `assume` imposes an obligation on the context to guarantee that the assumed condition holds
  - `assert` gives a guarantee to the context that the asserted condition holds

Assume What You Require, Assert What You Ensure

- Using this we bracket the assignment `val z: Z = x / y` in a pair consisting of an `assume` and an `assert` statement

```
assume(y != 0)
val z: Z = x / y
assert(z == x / y)
```

- Concerning its execution `assume` is not distinguishable from `assert` in Slang
- However, concerning the reasoning they are very different:
  - `assume` imposes an obligation on the context to guarantee that the assumed condition holds
  - `assert` gives a guarantee to the context that the asserted condition holds
- These are the two main concepts that determine a *contract*

Assume What You Require, Assert What You Ensure

- We have seen a use of the division operation in Example B

Assume What You Require, Assert What You Ensure

- We have seen a use of the division operation in Example B

```
// Caller:                        // Called:
assert(y != 0)
                                  assume(y != 0)
                                  val z: Z = x / y
                                  assert(z == x / y)
assume(z == x / y)
```

Assume What You Require, Assert What You Ensure

- We have seen a use of the division operation in Example B

```
// Caller:                    // Called:
assert(y != 0)
                              assume(y != 0)
                              val z: Z = x / y
                              assert(z == x / y)
assume(z == x / y)
```

- It is the obligation of the ‘Caller’ to ensure that the assumption of the ‘Called’ holds

Assume What You Require, Assert What You Ensure

- We have seen a use of the division operation in Example B

```
// Caller:                    // Called:
assert(y != 0)
                              assume(y != 0)
                              val z: Z = x / y
                              assert(z == x / y)
assume(z == x / y)
```

- It is the obligation of the ‘Caller’ to ensure that the assumption of the ‘Called’ holds
- In return the ‘Called’ guarantees that after its execution the assertion of the ‘Caller’ holds

Assume What You Require, Assert What You Ensure

- We have seen a use of the division operation in Example B

```
// Caller:                // Called:
assert(y != 0)
                          assume(y != 0)
                          val z: Z = x / y
                          assert(z == x / y)
assume(z == x / y)
```

- It is the obligation of the ‘Caller’ to ensure that the assumption of the ‘Called’ holds
- In return the ‘Called’ guarantees that after its execution the assertion of the ‘Caller’ holds
- This interplay between the ‘Caller’ and the ‘Called’ describes the essence of the contract between the two

Assume What You Require, Assert What You Ensure

- We have seen a use of the division operation in Example B

```
// Caller:                      // Called:
assert(y != 0)
                                assume(y != 0)
                                val z: Z = x / y
                                assert(z == x / y)
assume(z == x / y)
```

- It is the obligation of the ‘Caller’ to ensure that the assumption of the ‘Called’ holds
- In return the ‘Called’ guarantees that after its execution the assertion of the ‘Caller’ holds
- This interplay between the ‘Caller’ and the ‘Called’ describes the essence of the contract between the two
- It make reasoning about programs compositional, relying on the fulfilment of all contracts

Organisation

Course Topic and Scope
- Motivation
- Slang and Sireum/Logika
- Installing Sireum/Logika

Reasoning about Programs
- Example A: Observing a Logical Error
- Example B: Observing Undefined Behaviour

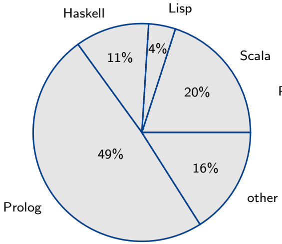
Methodology
- Formality
- Contracts and Reasoning Support
- A First Look at Contracts

The Programming Project

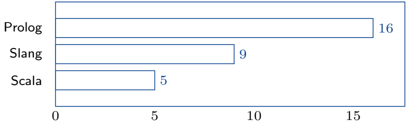
Summary

Programming Project Description

- A portable and scalable format, the so-called PSF, for graphics with text is needed that can be stored and sent via the Internet.
- The format should support drawing lines, rectangles, circles and text.
- The most important use of the format will be to draw charts representing data, e.g., bar charts and pie charts.



Popularity of Programming Languages



“What do you expect from life?”

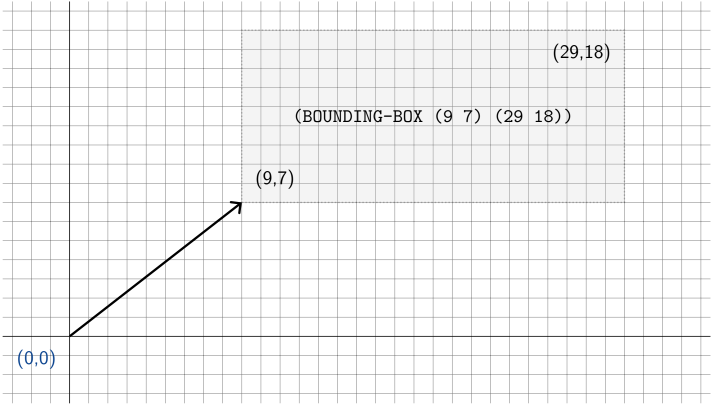
Domain Specific Language

- The PSF is defined in terms as a domain specific language with the commands listed below
- All commands must be implemented in the way specified

- `(LINE (x1 y1) (x2 y2))`: draw a line from `(x1 y1)` to `(x2 y2)`.
- `(RECTANGLE (x1 y1) (x2 y2))`: draw a rectangle with bottom left coordinate `(x1 y1)` and top right coordinate `(x2 y2)`.
- `(CIRCLE (x1 y1) r)`: draw a cicle with center `(x1 y1)` and radius `r`.
- `(TEXT-AT (x1 y1) t)`: draw the text `t` at `(x1 y1)`.
- `(BOUNDING-BOX (x1 y1) (x2 y2))`: the bounding box must be the first command.
- `(DRAW c g1 g2 g3 ... )`: draw the objects `g1`, `g2`, `g3`, ... in colour `c`, where `g1`, `g2`, `g3`, ... are any of the preceding commands like `(LINE (x1 y1) (x2 y2))`. (Outside a draw command the default colour black is used.)
- `(FILL c g)`: fill the object `g` with colour `c`, where `g` is any of the preceding commands.

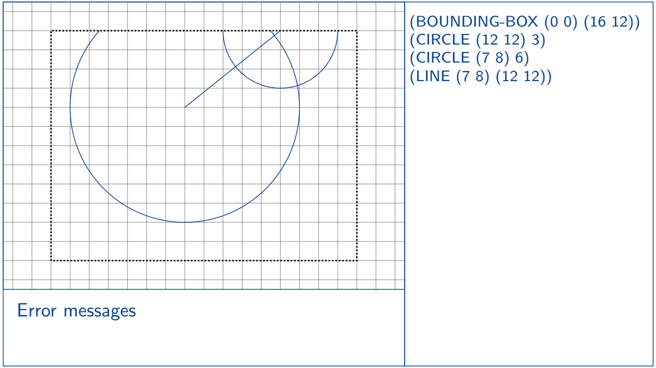
Drawing Plane and Bounding Box

- On the drawing plane a bounding box is set
- Each PSF document begins with a bounding box command
- All graphics drawing outside a bounding box is clipped



PSF IDE and GUI

- Below is a sketch of the expected GUI of the PSF IDE
- Keep the GUI simple to save effort



Organisation

Course Topic and Scope
- Motivation
- Slang and Sireum/Logika
- Installing Sireum/Logika

Reasoning about Programs
- Example A: Observing a Logical Error
- Example B: Observing Undefined Behaviour

Methodology
- Formality
- Contracts and Reasoning Support
- A First Look at Contracts

The Programming Project

Summary

Summary

Today we discussed

- Slang and Sireum/Logika
- Some concepts and principles of reasoning about programs in programming terms
- The programming project

In the coming lectures we discuss

- More Slang and Sireum/Logika!
- Slang constructs purely to support reasoning
- Automation in Sireum/Logika
- Proof and Test