Utrecht University

Concepts of Programming Language Design

Introduction

Gabriele Keller
Tom Smeding

Before we get started

Some admin issues

Before we get started

- Lectures & labs

  - will usually be recorded, but not streamed

  - usually, 2 x 45 min lecture, 45 min lab with exercises, questions, more on demand, lab not recorded

- Tuesday lecture & lab: 13:15 - 17:00

- Thursday lecture & lab: 10:00 - 12:45

- on demand: Thu 9:00 - 10:00 for extra help before assignment deadlines,

Course Website/Teams

- Course website:
  - https://utrechtuniversity.github.io/infomcpd/index.html
  - slides will be available on the website (will try and upload a draft version of the slides before the lecture)
  - course schedule and material
  - link to lecture notes and repo for lecture notes (you can report typos and such there)
- Blackboard
  - you can mostly ignore it
  - I'll use it to communicate grades for the assignments
  - for assignment 1 submissions

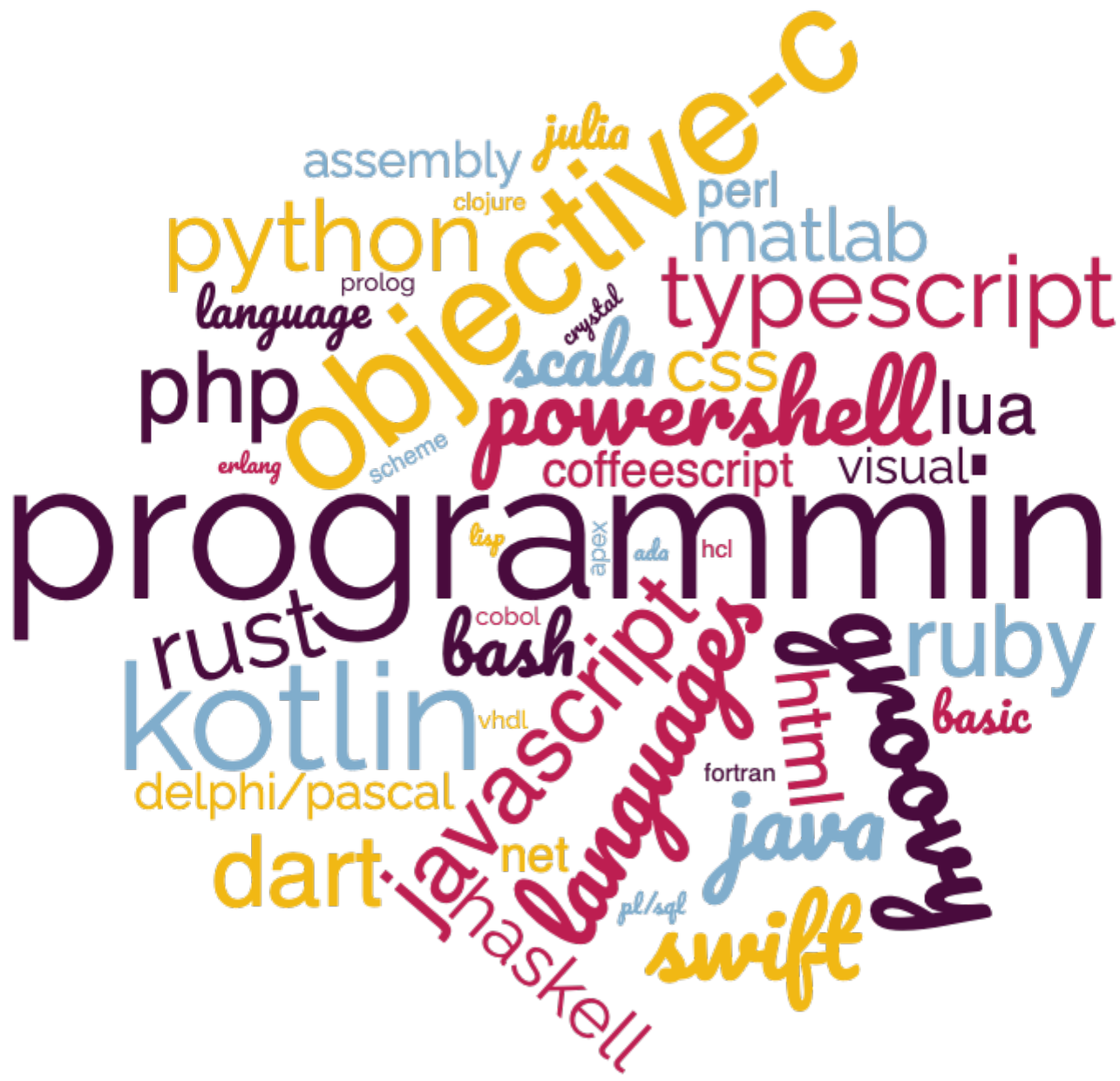
Course Website/Teams

- MS Teams for communication
  - you should have received an invitation to join the team, or use code `d7m6v3b`
  - questions about lecture, lecture notes, assignments via Teams
  - questions for me: direct msg on Teams
    - if you don't get a reply in a (working) day, feel free to ping again
- Lecture notes
  - link to the latest version on the course website
  - report typos, also requests for more explanations/examples, via GitLab
  - `https://git.science.uu.nl/g.k.keller/mcpd-lecture-notes/`
- Weekly exercises
  - weekly exercises for the lab

Course Organisation

- Assessment :
  - Practical part: counts for 50% of the mark
    - three small programming exercises (to help you learn Haskell, some concepts behind the lecture)
      - auto-marked, with feedback about style, better solutions in the lecture
      - programming assignment in Haskell, implementing one of the concepts, mainly auto marked
  - Final exam counts for 50% of the mark
    - one exam question will be about a language of your choice (out of a fixed set of languages, more on that soon)

Course Content



Utrecht University

So many languages, so little time!

Utrecht University

Which languages should we look at?

Utrecht University

Modern general purpose languages share many concepts

imperative	*OO*	*functional*
statically typed	**C#**	*polymorphic*
garbage collected		*strongly typed*

procedural *OO* *functional*

Python

duck typing *interpreted*

garbage collected *dynamically typed*

ctional

C#

statically typed *polymorphic*

garbage collected *strongly typed*

imperative *OO* *functional*

JavaScript

garbage collected *dynamically typed*

weakly typed

procedural

Python

interpreted

garbage collected *dynamically typed* *onal*

statically typed

C#

polymorphic

garbage collected

strongly typed

Utrecht University

imperative

OO

functional

JavaScript

procedural

ga

Py

garbage colle

generics/polymorphism

OO

functional

algebraic data types

Swift

statically typed

garbage collected

strongly typed

garbage collected

statically typed

C#

polymorphic

garbage collected

strongly typed

Utrecht University

imperative

OO

functional

JavaScript

procedural

generics/polymorphism

OO

functional

algebraic data types

Swift

statically typed

garbage collected

strongly typed

generics/polymorphism

type classes/traits

ownership types

Rust

statically typed

strongly typed

Utrecht University

object oriented

procedural/imperative

functional/declarative

dynamically typed

statically typed

strongly typed

weakly typed

pattern matching

lazy/strict

algebraic data types

generics/polymorphism

type classes/traits

overloading

ownership types

manual memory management

garbage collected

Utrecht University

We'll look at the

general concepts

programming languages are based on!

Utrecht University

Aim of this course

- The aim of this course is to

  - help you to understand and master new programming languages more quickly

  - understand the trade off between programming languages/features

  - reason (formally and informally) about programs and programming language features

  - give you the tools to understand the design and, to some extend, implementation of programming languages

  - provide a look ‘under the hood’ of high-level languages

Languages we use

To describe and prove properties of programming languages:

predicate logic, inductive definitions and inference rules (natural deduction)

Implementation language for projects, exercises, examples:

Haskell

We analyse and (in some cases) implement a variety of simple languages which showcase different PL concepts

Implementation of Programming Languages

```haskell
module Main where

import System
import System.IO
import Data.Bits
import Data.Word
import Data.Binary
import Data.Binary.Put

import qualified Data.ByteString.Lazy as DB

import Debug.Trace

data T = T Int Int Int Double
main = do
    -- width in pixels
    w <- getArgs >>= readIO . head
    let n    = w `div` 8
        -- width of a pixel in the complex plane
        m    = 2 / fromIntegral w
        coords = [T 1 0 y (fromIntegral y * m - 1) | y <- [0..w-1]]
        bs = concat $ concat $ map (worker w m n) coords
    putStrLn ("P4\n"++ show w ++ " " ++ show w)
    let bstr  = runPut $ mapM_ put bs
    DB.hPutStr stdout bstr


-- Worker computes one line of the image and sends it to the master
-- q - work queue
-- w - width in pixels
-- m - width of a pixel in the complex plane
-- n - width in bytes
worker w m n coord = do
    let p = unfold (next_x w m n)  coord
    return p


unfold :: (T -> Maybe (Word8,T)) -> T -> [Word8]
unfold f  x0 = go x0
  where
    -- x - coordinates

    go x = case f x of
        Just (w,y)          -> w : go  y
        Nothing             -> [0]


-- w - image width in pixels
-- iw - pixel width in the complex plane
-- bw - image width in bytes
next_x w iw bw (T bx x y ci)
    | bx == bw  = Nothing
    | otherwise = Just (loop_x w x 8 iw ci 0, T (bx+1) (x+8) y ci)
```

High-level language

assembly language

(dis)assembler

machine language

hardware

Implementation of Programming Languages

```haskell
module Main where

import System
import System.IO
import Data.Bits
import Data.Word
import Data.Binary
import Data.Binary.Put

import qualified Data.ByteString.Lazy as DB

import Debug.Trace

data T = T Int Int Int Double
main = do
    -- width in pixels
    w <- getArgs >>= readIO . head
    let n      = w `div` 8
        -- width of a pixel in the complex plane
        m  = 2 / fromIntegral w
        coords = [T 1 0 y (fromIntegral y * m - 1) | y <- [0..w-1]]
        bs = concat $ concat $ map (worker w m n) coords
    putStrLn ("P4\n"++ show w ++ " " ++ show w)
    let bstr  = runPut $ mapM_ put bs
    DB.hPutStr stdout bstr


-- Worker computes one line of the image and sends it to the master
-- q - work queue
-- w - width in pixels
-- m - width of a pixel in the complex plane
-- n - width in bytes
worker w m n coord = do
    let p = unfold (next_x w m n)  coord
    return p


unfold :: (T -> Maybe (Word8,T)) -> T -> [Word8]
unfold f  x0 = go x0
  where
    -- x - coordinates

    go x = case f x of
        Just (w,y)          -> w : go  y
        Nothing             -> [0]


-- w - image width in pixels
-- iw - pixel width in the complex plane
-- bw - image width in bytes
next_x w iw bw (T bx x y ci)
    | bx == bw  = Nothing
    | otherwise = Just (loop_x w x 8 iw ci 0, T (bx+1) (x+8) y ci)
```

High-level language

program as input

compiler

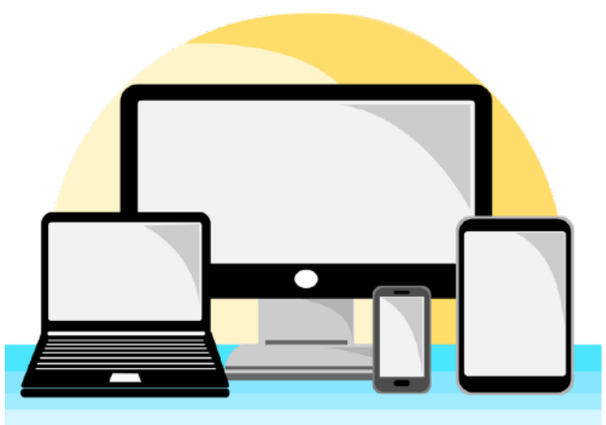
generates machine code

machine language

hardware

Implementation of Programming Languages

```haskell
module Main where

import System
import System.IO
import Data.Bits
import Data.Word
import Data.Binary
import Data.Binary.Put


import qualified Data.ByteString.Lazy as DB

import Debug.Trace

data T = T Int Int Int Double
main = do
    -- width in pixels
    w <- getArgs >>= readIO . head
    let n      = w `div` 8
        -- width of a pixel in the complex plane
        m  = 2 / fromIntegral w
        coords = [T 1 0 y (fromIntegral y * m - 1) | y <- [0..w-1]]
        bs = concat $ concat $ map (worker w m n) coords
    putStrLn ("P4\n"++ show w ++ " " ++ show w)
    let bstr  = runPut $ mapM_ put bs
    DB.hPutStr stdout bstr


-- Worker computes one line of the image and sends it to the master
-- q - work queue
-- w - width in pixels
-- m - width of a pixel in the complex plane
-- n - width in bytes
worker w m n coord = do
    let p = unfold (next_x w m n)  coord
    return p


unfold :: (T -> Maybe (Word8,T)) -> T -> [Word8]
unfold f  x0 = go x0
  where
    -- x - coordinates

    go x = case f x of
        Just (w,y)          -> w : go  y
        Nothing             -> [0]


-- w - image width in pixels
-- iw - pixel width in the complex plane
-- bw - image width in bytes
next_x w iw bw (T bx x y ci)
    | bx == bw  = Nothing
    | otherwise = Just (loop_x w x 8 iw ci 0, T (bx+1) (x+8) y ci)
```



High-level program

interpreter executes instructions of high-level program

interpreter

hardware

Implementation of Programming Languages

```haskell
module Main where

import System
import System.IO
import Data.Bits
import Data.Word
import Data.Binary
import Data.Binary.Put


import qualified Data.ByteString.Lazy as DB

import Debug.Trace

data T = T Int Int Int Double
main = do
    -- width in pixels
    w <- getArgs >>= readIO . head
    let n      = w `div` 8
        -- width of a pixel in the complex plane
        m  = 2 / fromIntegral w
        coords = [T 1 0 y (fromIntegral y * m - 1) | y <- [0..w-1]]
        bs = concat $ concat $ map (worker w m n) coords
    putStrLn ("P4\n"++ show w ++ " " ++ show w)
    let bstr  = runPut $ mapM_ put bs
    DB.hPutStr stdout bstr


-- Worker computes one line of the image and sends it to the master
-- q - work queue
-- w - width in pixels
-- m - width of a pixel in the complex plane
-- n - width in bytes
worker w m n coord = do
    let p = unfold (next_x w m n)  coord
    return p


unfold :: (T -> Maybe (Word8,T)) -> T -> [Word8]
unfold f  x0 = go x0
  where
    -- x - coordinates

    go x = case f x of
        Just (w,y)          -> w : go  y
        Nothing             -> [0]


-- w - image width in pixels
-- iw - pixel width in the complex plane
-- bw - image width in bytes
next_x w iw bw (T bx y ci)
    | bx == bw  = Nothing
    | otherwise = Just (loop_x w x 8 iw ci 0, T (bx+1) (x+8) y ci)
```

High-level program

program as input

compiler

generates intermediate representation

intermediate language

interpreter

hardware

just in time compilation

Implementation of Programming Languages

```haskell
module Main where

import System
import System.IO
import Data.Bits
import Data.Word
import Data.Binary
import Data.Binary.Put

import qualified Data.ByteString.Lazy as DB

import Debug.Trace

data T = T Int Int Int Double
main = do
    -- width in pixels
    w <- getArgs >>= readIO . head
    let n      = w `div` 8
        -- width of a pixel in the complex plane
        m  = 2 / fromIntegral w
        coords = [T 1 0 y (fromIntegral y * m - 1) | y <- [0..w-1]]
        bs = concat $ concat $ map (worker w m n) coords
    putStrLn ("P4\n"++ show w ++ " " ++ show w)
    let bstr  = runPut $ mapM_ put bs
    DB.hPutStr stdout bstr


-- Worker computes one line of the image and sends it to the master
-- q - work queue
-- w - width in pixels
-- m - width of a pixel in the complex plane
-- n - width in bytes
worker w m n coord = do
    let p = unfold (next_x w m n)  coord
    return p


unfold :: (T -> Maybe (Word8,T)) -> T -> [Word8]
unfold f  x0 = go x0
  where
    -- x - coordinates

    go x = case f x of
        Just (w,y)          -> w : go  y
        Nothing             -> [0]


-- w - image width in pixels
-- iw - pixel width in the complex plane
-- bw - image width in bytes
next_x w iw bw (T bx x y ci)
    | bx == bw  = Nothing
    | otherwise = Just (loop_x w x 8 iw ci 0, T (bx+1) (x+8) y ci)
```

High-level program

program as input

compiler

generates intermediate representation

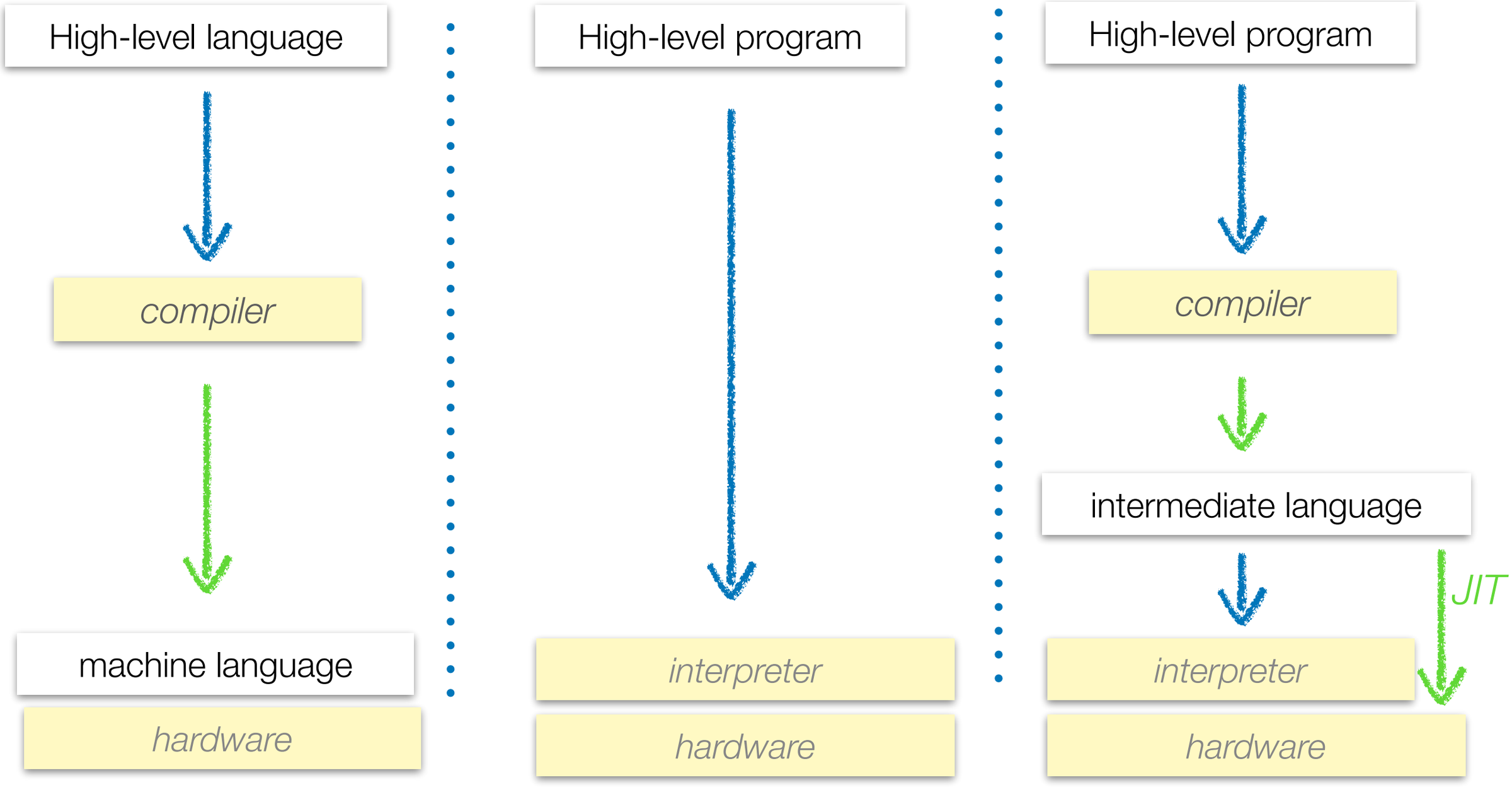
intermediate language

virtual machine

IR can be viewed as machine language of some virtual machine

Implementation of Programming Languages

What are the pros and cons of each approach?



Implementation of Programming Languages

- Examples for hybrid approaches using virtual machines:
  - Java, with Java Virtual Machine and Bytecode
  - .NET framework
  - LLVM IR in the LLVM framework
  - WebAssembly (WASM)

Implementation of Programming Languages

- Hybrid approach:
  - intermediate language, with some primitive data types, memory model, and interpreter
  - also called a virtual machine
- Abstract machines
  - like virtual machines, abstract machines are ‘imaginary machines’
  - usually used to specify operational semantics of a language, investigate some theoretical properties
  - not necessarily possible to implement (efficiently)

Topic overview

- Preliminaries:
  - predicate logic, inference rules, natural deduction
  - Haskell intro/revision
- Static and dynamic semantics
  - what is the static/dynamic semantics of a language?
  - how can we specify the semantics of a language
  - interaction between static and dynamic semantics
- Abstract machines
- Types
  - different flavours of polymorphism, static & dynamic typing, linear types
  - algebraic data types
  - reference types
- Core languages: MinHs, TinyC, Featherweight Java

This week

- We start with some theory revision
- introduction/revision of Haskell

Sue

Utrecht University

AI SINTERKLAAS HACKATHON

Join us for the third edition of the UU-Sue Student Hackathon! Get hands-on with the latest AI programming tools, connect, and code together for a good cause!

Open to new and experienced programmers

November 28, 2024 @ UU, 13:00-17:00
November 29, 2024 @ Sue, 10:00 - 17:00

Come along for...

- Learning the latest AI trends for software development
- Guest lectures from software engineers, and other IT professionals
- Coding for a good cause
- Free transport from/to the University - SUE
- Free Lunch and Dinner
- Free drinks

About

In this 2-day event, you will learn how AI tools, like Copilot and ChatGPT, are used in real-world software development, and work in small teams on a hands-on project guided by SUE engineers to build the necessary code and algorithms to help Sinterklaas deliver presents all around the country.

Enjoy networking opportunities throughout the event, and as a bonus, your contributions will support a charity focused on equal learning opportunities!

Register at

Utrecht University