Programming Languages – Summary

TABLE OF CONTENTS

1 INTRODUCTION

1.1 PROGRAMMING PARADIGMS

Def. Programming language:

> A programming language is a notational system for describing computation in a machine-readable and human-readable form. – Louden

Thesis of this course:

> A programming language is a tool for developing executable models for a class of problem domains.

Themes addressed in this course:

> Paradigms: How do different language paradigms support problem-solving?
>
> Foundations: What are the foundations of programming languages?
>
> Semantics: How can we understand the semantics of programming languages?

Generations of Programming Languages:

> 1 GL: machine codes
>
> 2 GL: symbolic assemblers
>
> 3 GL: (machine-independent) imperative languages – e.g. C
>
> 4 GL: domain specific application generators – e.g. SQL
>
> 5 GL: AI languages

Each generation is at a higher level of abstraction

Programming Paradigms:

> Imperative style: program = algorithms + data — good for decomposition
> Functional style: program = functions ∘ functions — good for reasoning
> Logic programming style: program = facts + rules — good for searching
> Object-oriented style: program = objects + messages — good for modelling
> …

Compiler vs Interpreter:

Compiler: Code -> Compiled Code -> Executed
Interpreter: Code -> Interpreted Line by line
-> In an Interpreter language, the source code must be present i.o to execute program, in compiled language, only compiled code must be present i.o. to execute program

1.2 Programming Language History

History go skra

2 Stack-based Programming

2.1 PostScript objects, types and stacks

PostScript is used to describe the appearance of text, graphical hapes, and sampled images on printed or displayed pages.

Code is usually generated from applications, rather than hand-coded.

Semantics:
program is sequence of tokens, representing typed objects, that is interpreted to manipulate the display and four stacks that represent the execution state of a PostScript program:

Operand stack:	holds (arbitrary) operands and results of PostScript operators
Dictionary stack:	holds only dictionaries where keys and values may be stored
Execution stack:	holds executable objects (e.g. procedures) in stages of execution
Graphics state stack:	keeps track of current coordinates etc.

Notice that Execution stack is mostly hidden from us, and is used by PostScript to manage running procedures.

Object types:

Literal objects:	pushed on operand stack > numbers, strings, arrays, procedures etc
Executable objects:	are interpreted > built-in operators, names bound to procedures
Simple Object Types:	Copied by values > Boolean, integer, name, …
Composite Object Types:	Copied by reference > array, dictionary, string, …

2.2 ARITHMETIC OPERATORS

The operand stack:

```
40 60 add 2 div
```

- 40: Add 40 to operand stack
- 60: Add 60 to operand stack
- Add: take two top elements from operand stack, add them, push result to operand stack
- 2: add 2 to operand stack
- Div: pop two top elements from operand stack, divide the bottom element by the top element, push result to operand stack
- (40 + 60) / 2 (Calculates average of 40 and 60)

Notice, that numbers are literal objects – see box above – and operators are executable objects

For a list of arithmetic operators: See Slides

2.3 GRAPHICS OPERATORS

Notice that Coordinates are measured in points.

Drawing a Box:

```
Newpath %clear current drawing path
100 100 moveto
100 200 lineto
200 200 lineto
200 100 lineto
100 100 lineto
10 setlinewidth % set width for drawing
stroke % draw along current path
showpage % display current page
```

Writing:

```
/Times-Roman findfont
18 scalefont
setfont
100 500 moveto
(Hello world) show
showpage
```

Notice, that /Times-Roman and (hello world) are literals, so are pushed on the stack, not executed.

2.4 PROCEDURES AND VAIRABLES

Variables and procedures are defined by binding names to literal or executable objects, e.g.:

```
/average { add 2 div } def
40 60 average % = 40 60 add 2 div
```

Notice, that after definition, average becomes an executable operator

PostScript programs usually consist of prologue and script:
prologue: application specific procedures (originally hand written)
script: usually generated, calls procedures and does stuff

Notice: Procedures should always clean up the state, meaning they should not leave unexpected objects on the operand stack: add 2 2 should consume two elements and leave one element on top of stack, the rest of the stack should be as before the execution and should not be manipulated.

2.5 ARRAYS AND DICTIONARIES

Dictionaries "override" each other, e.g. if I query a dictionary for a key, the toplevel dict is first considered, if key is not found, the dict underneath is considered and so on.

Rest See Slides

3 FUNCTIONAL PROGRAMMING

3.1 FUNCTIONAL VS. IMPERATIVE PROGRAMMING

Programs in functional languages have no explicit state, they are constructed entirely by composing expressions. A program is then a transformation from input data to output data.

Notice that often functional style resembles much more the typical mathematical definition than the imperative style.

Key features of pure functional languages:

1. Everything is a function
2. No variables or assignments
3. No loops, only recursive functions
4. Return value only depends on values of function parameters
5. Functions are first-class values

Higher-order functions: threat functions as first-class values, so can take functions as arguments and can yield functions as return values

Non-strict semantics: expressions are evaluated lazily, i.e. values are only computed as needed. This enables highly expressive language features such as infinite lists.

Haskell supports Higher-order functions and Non-strict semantics along other features.

3.2 Pattern Matching

Functions can be defined using patterns:

```
fac 0 = 1
fac n = n * fac (n - 1)
```

Functions can be definded using Guards:

```
fac n | n == 0          = 1
      | n >= 1          = n * fac (n – 1)
```

Evaluation order of unguarded patterns can be significant!

Lists are tuples of elements and lists of elements:

```
x:xs
```

is the list with x as head and xs as tail (where tail is a list aswell)

```
[1, 2, 3] = 1:2:3:[]
```

Lists are homogeneous – they can only contain elements of a single type.

More syntactic stuff: See slides

3.3 Referential Transparency

A function has the propery of *referential transparency* if its value depends only on the values of its parameters, e.g. if *f(x) + f(x) = 2 f(x)*

In pure functional languages, all functions are referentially transparent, therefor always yield the same result no matter how often they are called.

This is not necessarily true in other languages, e.g. in Java

```
this.incrementCounter()
```

does not necessarily always return the same value.

Evaluation of Expressions:
Expressions can be (formally evaluated by substituting arguments for formal parameters in function bodies:

```
fac 4
= if 4 == 0 then 1 else 4 * fac (4-1)
= 4 * fac (4-1)
= 4 * (if …)
…
```

In real functions that’s not how its implemented but this is useful for reasoning.

3.4 Lazy Evaluation

Lazy evaluation only evaluates expressions when they are actually needed. E.g.

```
sqr n = n * n
sqr (2 + 5)
= (2+5) * (2+5)
= 7 * 7
= 49
```

This allows e.g. for some functions to be evaluated even if they have incorrect or non-terminating arguments (if these arguments are not needed for the calculation). Additionally this allows for infinit stuff such as lists (as elements are only calculated if they are needed)

```
from n = n : from (n + 1) // from 100 = [100, 101, …]
```

3.5 Recursion

Recursive functions can be less efficient than loops because of high cost of procedure calls on most hardware.

A *tail recursive function* is a recursive function, that calls itself only as its last operation. These functions are beneficial, since the recursive call can be optimized away by modern compilers since it needs only a single run-time stack frame:

```
fact 5 -> fact 5 | fact 4 -> fact 5 | fact 4 | fact 3 -> … // this is bad

sfac 5 -> sfac 4 -> sfac 3 -> … // this is good since only one frame used (sfac is tail recursive)
```

A recursive function can be converted to a tail-recursive function by representing partial computations as explicit function parameters:

```
fact n =      if n == 0
              then 1
              else n * fact (n – 1)
              // Not tail recursive since last step is the multiplication!

sfac s n =    if n == 0
              then s
              else sfac (s * n) (n - 1)
              // Tail recursive since last step is recursive call!
```

Notice that naive recursion may result in unnecessary recalculations (e.g. in recursive fib calculation, same value is calculated multiple times!)

3.6 Higher Order and Curried Functions

Higher-order functions: threat functions as first-class values that can be composed to produce new functions. (functions can take functions as arguments)

First-class value: a value that can be passed as an argument, be returned from function (be assigned into a variable – does not exist in this context)

Second-class value: can only be passed as an argument

Third-class value: Not first class and not second class value.

Anonymous functions:

```
( \x -> x * x)
```

A *Curried function* takes its arguments one at a time, allowing it to be treated as a higher-order function.

This means that the function takes only one argument and returns a function that consumes the other arguments, e.g. a curried plus:

```
plus x = \y -> x + y
```

in Haskell:

```
plus x y = x + y // whereas plus (x y) = x + y is non curried !
```

Curried functions are useful because we can bind their arguments incrementally, e.g. an increment:

```
Inc = plus 1
```

Plus is only bound to one argument here, this returns a function that consumes another argument.

4 TYPES AND POLYMORPHISM

4.1 STATIC AND DYNAMIC TYPES

Notions of what a Type is

```
Type is a set of values -> very intuitive and natural

Type is a partial specification of behaviour -> e.g. Java Interface, might be useful
since we do not care about internal state of object but just about what it can do
```

Static Type: The type something has as it is declared in the source code
A language is *statically typed* if it is always possible to determine the (static) type of an expression based on the program text alone.

Dynamic Type: The type something has at Runtime
A language is *dynamically typed* if only values have fixed type. Variables and parameters may take on different types at run-time, and must be checked immediately before they are used.

A language is *strongly typed* if it is impossible to perform an operation on the wrong kind of object.

```
Applet myApplet = new GameApplet()
```

myApplet has static type Applet, since the variable is declared to be of type applet. It's dynamic type is GameApplet, since a GameApplet object is inside the variable.

The value (GameApplet()) has static and dynamic type GameApplet.

Usually a static type system prohibits you from running code if it can not validate the type correctness. A program might be semantically okay but it might not be okay in terms of static typing. In this case, you e.g. need to downcast.

Kinds of types:

- Primitive types: Booleans, integers, …
- Composite types: functions, lists, …
- User-defined types: Enums, recursive types, objects, …

4.2 Type Completeness

Type completeness principle:

> No operation should be arbitrarily restricted in the types of values involved.

WHAT DOES THIS MEAN?

4.3 Types in Haskell

Functions:
fact :: Int -> Int
plus :: Int -> Int -> Int (if plus is curried)

Lists:

[1] :: [Int]
Remember that all list elements must be of the same type!

Tuples:

('a', False) :: (Char, Bool)

User Data Types:

```
Data DatatypeName a1 … an = constr 1 | … | constr m
```

Where constructors may be
named constructors:

```
Name type1 … typek
```

Binary constructors (i.e. anything starting with ":"):

```
type1 BINOP type2
```

e.g. Enumaration Type:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
whatShallIDo Sun = "relax"
whatShallIDo Sat = "shopping"
whatShallIDo _ = "work"
```

Notice that Day has seven constructors without argument, for finding out which value a day is, we just pattern match.

e.g. Union Type:

```
data Temp = Centigrade Float | Fahrenheit Float
freezing :: Temp -> Bool
freezing (Centigrade temp) = temp <= 0.0
freezing (Fahrenheit temp) = temp <= 32.0
```

Notice that we deconstruct the value by patternmatching to gain access to information hidden inside.

e.g. Recursive Type:

```
data Tree a = Lf a | Tree a :^: Tree a
mytree = (Lf 12 :^: (Lf 23 :^: Lf 13)) :^: Lf 10
-> mytree :: Tree Int
```

Notice: :^: is our binary constructor, Lf is our other constructor

4.4 MONOMORPHIC AND POLYMORPHIC TYPES

Monomorphic type system: every constant variable, parameter and function result has a unique type e.g. C and Pascal.
This is good for type-checking but bad for writing generic code, e.g. in Pascal it's impossible to write a generic sort procedure.

Polymorphic type system: A type system that is not Monomorphic.
E.g. Java
This is good for writing generic code.

A *polymorphic function* accepts arguments of different types.
e.g.

```
Length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

Kinds of Polymorphism:

- Universal polymorphism:
  - Parametric: polymorphic map function in Haskell; Generic Classes in Java
  - Inclusion: subtyping – graphic objects
- Ad Hoc polymorphism:
  - Overloading: + applies to both integers and reals (but there are two separate definitions of +, not a single one!)
  - Coercion: one type can be used where another one is expected and v.v. e.g. String result = 13 + " is a number but the result is a string"

4.5 Hindley-Milner Type Inference

We can infer the type of many expressions by simply examining their structure.

```
Length [] = 0
length (x:xs) = 1 + length xs

length :: a -> b
length :: [c] -> Int
```

Here we cannot further refine the type, thus we're done.

Composing polymorphic types:

We can deduce types when using polymorphic functions by binding type variable to concrete types:

```
length :: [a] -> Int
map :: (a -> b) -> [a] -> [b]
// In this case:
map length :: [[a]] -> [Int]
["Hello", "World"] :: [[Char]]
map length ["Hello", "World"] :: [Int]
```

The *Hindley-Milner type inference* algorithm automatically determines the types of many polymorphic functions. It only works with parametric polymorphism, not subtype polymorphism, e.g. its unsuited for java, but well suited for Haskell.

Type Specializations:

We can add a more specific type to a function:

```
idInt :: Int -> Int
idInt x = x
// then idInt :: a -> a is no longer true, instead we manually set idInt :: Int -> Int
```

4.6 OVERLOADING

Coercion vs overloading:

```
3 + 4
3.0 + 4
3 + 4.0
3.0 + 4.0
```

Are there four overloaded + functions; two with coercion to real if one arg is int; or one with coercion to real? We cannot know.

Haskell supports overloading with *type classes*

```
Class Eq a where
(==), (/=) :: a -> a -> Bool
x /= y = not (x == y)
```

Such a class must be instanciated to be used:

```
Instance Eq Bool where
True == True = True
False == False = True
_ == _ = False
```

Notice that a type class has nothing to do with a class in OOP. Its more like an interface. All provided overloaded operators must be implemented by a datatype.

For each overloaded instance, a separate definition must be given, like above.

You can use implication to specigy derived type classes:

```
Instance Eq a => Eq [a] where
[] == [] = True
[] == (y:ys) = False
(x:xs) == [] = False
(x:xs) == (y:ys) = x == y && xs == ys
```

It's not possible to automatically provide e.g. equality for all types of values, because we do not know what equality means for a specific type, e.g. equal things can be represented in different ways, e.g. a in a set [0, 1] == [1, 0], but for lists this is wrong.

Equality for Functions:

Not possible to determine, because equality of functions is undecidable in general! (Halting Problem!)

5 LAMBDA CALCULUS

5.1 WHAT IS COMPUTABILITY? – CHURCH'S THESIS

Churchs Thesis:

> Effectively computable functions [from positive ints to positive ints] are just those definable in the lambda calculus.

Equivalently:

> It is not possible to build a machine that is more powerful than a Turing machine.

Cannot be proven since "effectively computable" is an intuitive notion. It can only be refuted by giving a counter-example – a more powerful machine than a Turing Machine.

Uncomputability: A problem that cannot be solved by any Turing machine in finite time.

If Churchs Thesis is true, than an uncomputable problem cannot be solved by a real computer.

E.g. the Halting problem can not be solved by a Turing Machine.

What is a function?

- Extensional view: f:A -> B is a subset of A x B s.t.
  - For each a in A, there is (a, b) in f (f(a) is defined)
  - If (a, b_1) in f and (a, b_2) in f, then $b_1 = b_2$ (f(a) is unique)
- Intensional view: f:A -> B is an abstraction λx.e where x is a variable name and e is an expression, such that when a value a is substituted for x in e, then this expression (e.g. f(a)) evaluates to some unique value b. – This means it's a specification of how to transform an input to an output.

5.2 LAMBDA CALCULUS – OPERATIONAL SEMANTICS

Syntax:

```
e ::= x //a variable
   | λx.e //an abstraction / function
   | e1e2 //a (function) application
```

Parsing lambda expressions:

- Lambda extends as far as possible to the right
  λf.x y = λf.(x y)
- Application is left-associative
  x y z = (x y) z
- Multiple lambdas may be suppressed
  λf g.x = λf. λg.x

(Operational) Semantics ([y/x] means that all x are replaced by y in the corresponding section, e.g. the body of an abstraction.):

- λx.e = λy.[y/x]e – alpha conversion (renaming) – notice that y must not exist as a free variable in the same scope.
- $(\lambda x.e_1)e_2 = [e_2 / x]e_1$ – beta reduction – notice that we must avoid name capturing
- λx.ex = e – eta reduction – if x is not a free variable in e

Example:

- (λx.x) (λx.x) = (λx.x) – if [(λx.x) / x] – This is a beta reduction

Bound / Free variables:

A variable x is bound by λ in expression λx.e
A variable is free if it is not bound

Expressions without free variables are *closed* (AKA a *combinator*)
Otherwise they are *open*.

5.3 The Church-Rosser Property

Why macro expansion is wrong

(λxy.xy)y -> [y/x] (λy.xy) (beta reduction)
= (λy.yy) – however this is clearly wrong, since y is already bound in the lambda, thus we cannot directly substitute y for x.
We must thus define substitution carefully to avoid *name capture*!
(This means we rename conflicting variables, defined inside the lambda, in case of naming conflicts.)

e.g.

```
(λx.(( λy.x)( λx.x))x)y = ((λz.y)( λx.x))y
```

What we do here is renaming the conflicting y to z and then we do a beta reduction.

A lambda expression is in *Normal Form* if it can no longer be reduced by beta or eta reduction rules. Notice that not all expressions have normal forms, e.g. (λx.xx) (λx.xx)

Reducing a lambda expression to a normal form is analogous to a turing machine halting or a program terminating.

Applicative-order reduction: Arguments of function are evaluated, then function is evaluated

Normal-order reduction: expressions are only evaluated when they are needed (arguments of functions are evaluated during function call) -> Lazy evaluation

The *Church-Rosser Property*:

If an expression can be evaluated, it can be evaluated by consistently using normal-order evaluation. If an expression can be evaluated in several different orders, then all of these evaluations yield the same result

- Evaluation order does not matter in lambda calculus (however remember that applications still happen left to right!)

5.4 MODELLING BASIC PROGRAMMING CONSTRUCTS

Non termination:

Applicative order reduction may not terminate, even if a normal form exists!, e.g.

```
(λ.y)(( λx.xx)( λx.xx))
```

Notice, the normal form is y, but in applicative order we will never reach the normal form.

Notice, that all lambda expressions with multiple variables are curried functions, since a lambda abstraction only binds a single variable.

Rest: See Slides

6 FIXED POINTS

6.1 REPRESENTING NUMBERS

See Slides

6.2 RECURSION AND THE FIXED-POINT COMBINATOR

If we work with numbers, we could try to "define" e.g. plus like this:

```
plus = λ n m . iszero n m (plus (pred n) (succ m))
```

However this is not a definition since we use plus before it is defined -> plus is free in the "definition".

We can obtain a closed expression by abstracting over plus:

```
rplus = λ plus n m . iszero n m (plus (pred n) (succ m))
```

rplus takes the actual plus as its argument and returns that function in terms of itself.

In other words, if fplus is the function we want, then:

rplus fplus = fplus -> we are searching for a fixed point of rplus.

A *fixed point* of a function f is a value p such that f p = p, e.g. fact 1 = 1

```
Fixed Point Theorem:
Every lambda expression e has a fixed point p, such that (e p) = p
```

This fixed point is the magical Y combinator:

$$Y = \lambda f.\ (\lambda x\ .\ f\ (x\ x))(\ \lambda x\ .\ f(x\ x))$$

Then for every expression e:
p = Y e = e p

Or

For all expressions e: Y e = e (Y e)

We seek a fixed point of rplus:

By the Fixed Point Theroem, we simply define:
plus = Y rplus, since this guarantees that rplus plus = plus as desired.

Notis that this plus is different from the plus bound by the lambda in rplus!

Unfolding Recursive Lambda Expressions:

SEE SLIDE 16 – THIS IS IMPORTANT!

6.3 THE TYPED LAMBDA CALCULUS

Decorates terms with type annotations

Syntax:

$e ::= x^{\tau} \mid e_1^{\tau 2 \to \tau 1}\ e_2^{\tau 2} \mid (\lambda\ x^{\tau 2}.e^{\tau 1})^{\tau 2 \to \tau 1}$

Semantics same as in regular lambda calculus but with type annotations as shown above.

6.4 THE POLYMORPHIC LAMBDA CALCULUS

Polymorphic functions like “map” or “length” cannot be typed in the typed lambda calculus.
We need *type variables* to capture polymorphism:

$(\lambda\ x^{v}.e^{\tau 1})\ e^{\tau 2}$

Polimorphism and self application:

Even polymorphic lambda calculus is not powerful enough to express certain lambda terms.

E.g. there is no reasonable type for the Y combinator, since there is no reasonable type for an expression like x x. For this reason you can not program the Y combinator in Haskell.
This makes recursion with the Y combinator impossible in most cases, thus most languages provide support for recursive functions with a def keyword (afterwards μ is used for this), which avoids using Y.

6.5 OTHER CALCULI

Not Important

7 Introduction to Denotational Semantics

7.1 Syntax and Semantics

Three main characteristics of programming languages:

- Syntax – Appearance and structure of programs
- Semantics – Meaning of programs
    - Static Semantics – which syntactically valid programs are semantically valid (i.e. which are type correct)
    - Dynamic Semantics – how to interpret meaning of valid programs
- Pragmatics – What is the usability? How easy is something to implement? What kind of applications does it suit?

Semantic Specifications:

- A precise standard for a computer implementation – How should language be implemented?
- User documentation – what is the meaning of a program?
- Tool for design and analysis – how can definition be tuned, s.t. it can be implemented efficiently?
- Input to compiler generator – How can reference implementation be obtained from spec?

Methods for Specifying Semantics:

Operational Semantics

- $[\![\text{program}]\!] = \text{abstract machine program}$
- Simple to implement but hard to reason about

Axiomatic Semantics:

- $[\![\text{program}]\!] = \text{set of properties}$
- Good for proving theorems about programs but somewhat distant from implementation

Denotational Semantics (We focus on this):

- $[\![\text{program}]\!] = \text{mathematical denotation (typically functions)}$
- Good for reasoning but not always easy to find suitable semantic domains

Structured Operational Semantics:

- $[\![\text{program}]\!] = \text{transition system (definded using inference rules)}$
- Good for concurrency + non-determinism but hard to reason about equivalence.

Semantic brackets usage:
$[\![$ <program fragment> $]\!]$ = <math. object>

7.2 SEMANTICS OF EXPRESSIONS

Concrete and Abstract Syntax

Abstract Syntax is compact but ambiguous:

```
Expr ::= Num | Expr Op Expr
Op ::= + | - | * | /
```

Concrete Syntax is unambiguous but verbose :

```
Expr ::= Expr LowOp Term | Term
Term ::= Term HighOp Factor | Factor
Factor ::= Num | (Expr)
LowOp ::= + | -
HighOp ::= * | /
```

Notice: In these examples Concrete Syntax specifies e.g. that * and / must be executed before + and - , this is not specified in abstract syntax, there we only specify what is syntactically valid

- For parsing we need concrete syntax, for semantic specifications, abstract syntax is enough. In the semantic specification we take a parsed abstract syntax tree as input, thus the information that + / - are lowOp is already in there, thus we do not need it that information anymore.

Semantic Functions:

We can specify the semantics of some abstract syntax like this:

```
E: Expression -> Int
E ⟦Num⟧ = Num
E ⟦Expr + Expr⟧ = E ⟦Expr⟧ + E ⟦Expr⟧
E ⟦Expr – Expr⟧ = E ⟦Expr⟧ - E ⟦Expr⟧
E ⟦Expr * Expr⟧ = E ⟦Expr⟧ * E ⟦Expr⟧
E ⟦Expr / Expr⟧ = E ⟦Expr⟧ / E ⟦Expr⟧
```

More examples see Slides

Semantic Domains

To define semantic mappings of programs to mathematical denotations, domains must be precisely specified.

e.g.

```
data Bool = True| False
False && x = False
…
```

We want to interpret abstract syntax trees with our semantics, this means we always assume that the code is already parsed into a tree determining what is executed when.

Notice that denotational semantics can directly be implemented in functional languages like Haskell – See Slide 16.

7.3 Semantics of Assignment

In Hasekell we can model Assignments as mapping from identifiers to values (= A store)

The Key Idea is to have an update function, where you return a function recursively if the current key does not match the searched key.

Details See Slide 21

7.4 Other Issues

Practical:

- Errors and non-termination -> special error value in semantic domain.
- Interactive input
- Dynamic typing
- …

Theoretical:

- Semantics of recursive functions?
- Model concurrency and non-determinism?
- …

8 Objects and Prototypes

8.1 Class- vs. prototype-based languages

Class-based:

- Methods shared, define common properties
- Inheritance
- Instance has exact properties and behavior definded by class
- Structure (typically) can't be changed at runtime

Prototype-based:

- No classes, only objects (Examples first, not abstraction first)
- Objects define own properties and methods
- Objects delegate to prototype(s)
- Any object can be prototype of other object

- Prototype-based languages unigy objects and classes

8.2 Objects, properties and methods

This is just JS Syntax which is easy -> See Slides if unsure.

8.3 Delegation

Objects can delegate to their prototype:

```
var counter2 = Object.create(counter1); // counter1 is prototype of counter1
counter2.val = 0;
counter2.name = “counter2”; //overwrite stuff
counter2.inc(); // calls inc() in the prototype since not overwritten, but uses
counter2.val rather than counter1.val!
```

Notice in this example ‘inc’ in counter2 === true but counter2.hasOwnProperty(‘inc’) === false!

8.4 Constructors

Constructor = functions that are used together with ‘new’ notice that new binds the created object to ‘this’, by default the return value is the object that is created.

```
function Counter(name) {
  this.val = 0;
  this.name = name;
}
var counter3 = new Counter(“counter3”);
counter3.val; //-> 0
counter3.name; //-> “counter3”
```

Notice that calling the Constructor without new is fatal, then no object is created, but instead properties of ‘this’ are manipulated (and you don’t know what this is atm).

Prototype of Constructors:
instead of adding same method to all objects, we can add it to the constructors prototype:

```
Counter.prototype.inc = counter1.inc;
```

Then ‘counter3.inc()’ works as expected.

You can call a function with more arguments than it is formally declared to accept:

```
function example(argument) {
   return arguments[1]
}
example(1, 2) // -> 2
```

8.5 Closures

Variables and functions can be scoped, e.g. “made private”.
there is function and global scope, but no block-level scope!

Functions are executed in the scope in which they are created, not from where they are called, this allows for closures (Inner functions):

```
function f(x) {
  var y = 1 ;
  return function() { return x + y };
}
closure = f(2)
var y = 99;
closure(); //-> 3
```

A *closure* is a function whose free variables are bound by an associated environment.

```
var counter = (function(name) {
  var val = 0;
  var name = name;
  return {
    inc: function() {val++;},
    get val() {return val},
    get name() {return name}
  };
})('counter4')

counter.val // readonly
counter.name // readonly
```

8.6 Snakes and Ladders with Prototypes

See Slides (Not very important)

8.7 The Good, the Bad and the Ugly

The Good:

- Object literals, JSON
- Object.create()
- Dynamic object extension
- First-class functions
- Closures

The Bad:

- Global variables
  - This may bind to global object
- Undeclared variables are new globals
- No nested block scopes
- Arrays are not real arrays (Slow + inefficient) + generally arrays are shit
- Typeof returns strange results sometimes
- Equality (==) is not symmetric and has issues with coercion
- Constructor call without new = fatal

The Ugly

- No standard for setting prototype of an object
  - __proto__ is browser specific
- Single prototype chain -> No multiple delegation
- New class style programming is only syntactic sugar -> weird side effects
  - Use Object.create() instead!

9 OBJECTS, TYPES AND CLASSES

9.1 OBJECTS AND TYPE COMPATIBILITY

9.1.1 The Principle of Substitutability

Subclassing != Subtyping != is-a relation:

Subclassing = incremental modification
Subtyping = substitutability
Is-a relation = specialization

The principle of Substitutability:

> An instance of a subtype can always be used in any context in which an instance of a supertype was expected.

What we expect here:
A client has expectations of a component, a supplier provides a component that satisfies these expectations. This is a compatible component.

Syntactic compatibility: The component provides all the expected operations (type names, function signatures, interfaces, …)

Semantic compatibility: The component's operations behave in the expected way (state semantics, logical axioms, proofs, …)

Liskovs substitutability principle:

> Let q(x) be a property provable about objects x of type T.
> Then q(y) should be true for objects y of type S, where S is a subtype of T.

9.1.2 Types and Polymorphism

Kinds of Type Compatibility:

- Exact Correspondence: component is identical in type and behaviour exactly matches expectations when called through the interface
- Subtyping: component is more specific type, but behaves exactly like the more general expectations when called through the interface
- Subclassing: component is more specific type and behaves in ways that exceed the more general expectations when called through the interface.

There are different kinds of types – we focus on *Syntactic and existential abstract types* (types as sets of function signatures), since this is the most practical approach.

9.1.3 Type rules and type-checking

Type Rules:

$\frac{a,b}{c}$ – This means: if a and b then c, e.g.

$\frac{n:N,m:M}{<n,m>:N\ x\ M}$ – "if n is of Type N, and m is of type M, then tuple <n, m> is of type N x M

In the lecture there are many such rules introduced, I only list those that are not that easy to read out of the box:

- $\frac{x:D \vdash e:C}{\lambda x.e:D \to C}$ - Function Introduction (⊢ means "if you can conclude from left, that right)

Our goal is to have type rules, and conclude types of other stuff using our type rules.
E.g. if we have a function make-point that takes a tuple of integers we want to conclude that the function is of type Integer x Integer -> Point, however we don't know what a Point type is, thus we must look closer into objects

9.1.4 Object encodings and recursion

What is an object?

- Existential object encoding -> Object as data abstraction
  "There is some (hidden) representation that makes an Interface work as it should
  + models private state
  + no recursive types
  - clumsy invocation
- Functional object encoding -> Object as functional closure [WE FOCUS ON THIS]
  "a *Point*(or whatever type you want) is a record of methods, some of which manipulate *Point* objects"
  This implies recursive typing, since e.g. equal: pnt -> Boolean, thus we need μ/let from lambda calculus
  + models private state (through closures)
  + direct interpretation of methods
  - requires fixpoints
- Object calculus -> Object as self-bound record

9.2 OBJECTS AND SUBTYPING

9.2.1 Subtypes, Covariance and Contravariance

Types as Sets:

x ∈ X -> x: X
Y ⊆ X -> Y < : X
E.g. john : Student -> john: Person

This view is intuitive, but leads to conceptual problems

Function subtyping:
f: A -> B
g: C -> D

When can g be safely substituted for f?

Covariant types:
C ⊆ A and D ⊆ B
if a client applies g to a value in A \ C, there will be a runtime error.

Contravariant types:
A ⊆ C and D ⊆ B
guarantees, that a client will not receive unexpected results.

Definition of Subtype:

$$\frac{A <: C, D <: B}{C \to D <: A \to B}$$

This is a nice clean rule, but it violates our modeling principles, we usuall want to specialize in domain and co-domain.

Overloading:
If we overload a method with a covariant type, this looks very nice (and is useful most of the time), however the static type decides which function is called, which can lead to problems.
e.g. if we have:

```
Point p1 = new HotPoint(x, y);
Point p2 = new HotPoint(x, y);
p1.compare(p2);
```

Then the compare method of Point is called and NOT the one from HotPoint, which might have unexpected side effects.

9.2.2 Checking subtypes

Record Extension: A record (object) with more fields, can be safely substituted for one with fewer fields.

Record Overriding: The type of a record is a subtype of another record, if each of its fields is a subtype of the same field of the second. (This is however not very useful in practice since we want specialization, not subtyping).

9.2.3 Classes as families of types

The problem with recursive closure:
Suppose:

```
Animal = μ σ. {..., mate: σ → σ , ...}
Animal.mate: Animal → Animal

Dog.mate: Dog → Dog
Cat.mate: Cat → Cat
// Covariance breaks subtyping
```

```
Dog = μ σ. {…, mate: Animal → σ , …}
Dog.mate: Animal → Dog
// Contravariance preserves subtyping but breaks nature!
```

If we do something like this, then we do not do subtyping but subclassing!

The Problem of Type-loss:
Consider

```
Number = μ σ. {plus: σ → σ}
       = {plus: Number → Number}

Integer = μ σ. Number ∪ {minus: σ → σ}
        = {plus: Number → Number,
           minus: Integer → Integer}

i, j, k: Integer
i.plus(j).minus(k) // fails to typecheck
```

Classes as Type Generators:

```
GenNumber = λ σ. { plus : σ→σ }
GenNumber[Number] = { plus : Number → Number }
GenNumber[Integer] = { plus : Integer → Integer }

Integer <: Number // That's not the case
Integer <: GenNumber[Integer] //That's the case, since no more than a record
extension
```

- ➔ Classes are generators of types, since like this we can have proper subtyping.
  e.g. the class of Numbers represents the family of types which are subtypes of GenNumber applied to themselves.
- ➔ Generally a class represents a family of types, e.g. an Animal class represents all Animals, however there is not necessarily a subtype relationship
- ➔ Consequently Inheritance is subclassing without subtyping, e.g. GenNumber is the Union of GenInteger and additional stuff, Integer then is a subtype of GenInteger.

10 Logic Programming

10.1 Facts and Rules

In Logic Programming:
Program = Facts + Rules

Fact = named relation between objects:

```
parent(charles, elizabeth).
```

Rules = relations that can be inferred from other relations:

```
mother(X, M) :- parent(X, M), female(M).
```

Rules (and facts in a trivial way, with 'if true') are instances of *Horn clauses*
A Horn clause is something like:

```
H if B0 and B1 … Bn
```

H is head, after if is body

Questions / Queries = statements that can be answered using facts and rules:

```
?- mother(charles, M).
-> M = elizabeth
```

Closed world assumption:

Anything that cannot be inferred with given data is assumed false.

10.2 Resolution and Unification

Query resultion by:

- **Matching** (sub)goals against facts or rules
  If head of a rule matches a subgoal, we can replace subgoal with body of the rule.
  e.g.
  mother(diana, C) -> parent(diana, c), female(diana) -> …
- **Unifying** free **variables** with terms (NOT assignment)
  constant unifies only with same constant, e.g. elizabeth = Elizabeth
  free variable unifies with everything, e.g. Y = elizabeth
  terms unify, if same functor name, same number of arguments, and arguments can be unified recursively:
  parent(elizabeth, X) = parent(Y, Z)
  This is used to find actual solutions to a query, e.g.
  if I query mother(diana, C), and I have matched subgoals, I get:
  parent(diana,C),female(diana). By unification, I find e.g. C=william

- **Backtracking** when (sub)goal matching fails – notice that subgoals are substituted left to right and clauses are tried top-to-bottom

Comparison: a == b, if a and b strictly identical – functors match, number of arguments matches, free variables are shared (same name in scope or explicitly unified)

e.g.

```
P = diana.
parent(P, harry) == parent(diana, harry).
```

10.3 Recursion, Functions and Arithmetic

Recursions are defined in the obvious way:

```
ancestor(A, D) :- parent(A, D).
ancestor(A, D) :- parent(P, D), ancestor(A, P).
```

Keep in mind that clauses are evaluated top, to bottom, meaning base case must come first, recursive goal last!

Search can be controlled using 'fail/0' and Cuts

Cut (!) says "stop backtracking from here on"
Green cut: does not change semantics of program, just eliminates useless searching:

```
max(X, Y, X) :- X > Y, !. % mutually exclusive cases, optmization
max(X, Y, Y) :- X =< Y.
```

Red cut : changes semantics of program, incorrect results if you remove the cut!

```
max(X, Y, X) :- X > Y, !. % not really mutually exclusive:
max(_, Y, Y). % this clause will succeed if comarison before cut fails
```

No built in 'not' operator, but:

```
not(X) :- X, !, fail. % if X succeeds, we fail
not(_). % if X fails, we succeed
```

Notice that this is a red cut!

Arithmetics:

```
X is 3 + 4 // function from right hand side to left hand side!
7 + 4 =\= 10 + 2
2 + 2 =:= 3 + 1
```

Functions:

```
fact(0, 1).
fact(N, F) :- N > 0,
              N1 is N - 1,
              fact(N1,F1),
              F is N * F1.
```

10.4 Lists and other Structures

List construction:

Formal	Cons Pair	Element
.(a, [])	[a | []]	[a]
.(a, .(b, []))	[a | [b | []]]	[a, b]

As previously seen in Haskell, Lists consist of a head and a Tail. Free variables are allowed.

Example use of lists:

```
in (X, [X | _]).
in (X, [_ | L] :- in (X, L).
```

11 Applications of Logic Programming

11.1 Efficiency in computations

You can return sets of answers collected with backtracking, e.g.
setof/3:

```
setof(C, parent(P,C), S).
```

About how setoff works:

```
setof(A, B, C).
```

A : The template, which means if a variable is listed here, and then used in the predicate, the matches for this variable are listed in the solutions
B : The predicate, what must be satisfied i.o to be an element of the set?
C : In here, the Solutions are placed

Examples:

```
setof(C, parent(elizabeth, C), S). // -> returns all children of Elizabeth
setof(C, parent(C, william), S). // -> returns the parents of William
setof([A, B] parent(A, B), S). // -> returns a list of lists [A, B] where A is parent of B.
```

bagof/3:
like setoff/3, but results are NOT sorted.

findall/3:
similar to bagof/3

findnsols/4:
like findall/3 but only n solutions are returned

Higher Order Predicates:

There are predicates, that accept other predicates as arguments:

```
maplist(inc, [1,2], L). // -> L = [2,3] if inc is implemented as expected
maplist(sum, [1,2], [3,4], L). // -> L = [4,6]
filter(even, [1,2], L). // -> L = [2]
partition(even, [1,2], L, R). // -> L = [2], R = [1]
partition(smaller(2), [1,2], L, R). // -> L = [1], R = [2] with a smaller(Comperator,
X)
```

11.2 DATALOG

Logic Programming + Relational Databases = Deductive Databases
Deductive Databases can make inference of additional facts based on relations already present in the database.
Data representation: relations (based on Horn clauses)
Data Query: with Datalog language.

Datalog Differences to Prolog:

- Facts can be negated (e.g. 'not Q(x,y,_)' with Q being a Fact)
- Clause order does not matter
- No cut operator
- Something like person(name(elizabeth)) is not legal

11.3 CLP AND NUMERIC COMPUTATIONS

CLP = Constraint Logic Programming

Used for answering Questions with constraints attached, e.g. in AI

E.g. route planning with time budget

CLP thus includes constraint satisfaction.
CLP = Prolog + Solver (for a given domain, e.g. real numbers -> CLP(R))

e.g. Dot Product:

```
prod([], [], Result) :- Result .=. 0.
prod([X|Xs], [Y|Ys], Result) :- Result .=. X * Y + Rest,
                                 prod(Xs, Ys, Rest).
```

11.4 ASSERTIONS AND PROGRAM VERIFICATION

Does not seem to be important to me. -> See Slides

11.5 Natural language parsing with DCGs

We want to check wether a sentence, represented as a list in Prolog, is grammatically correct.

e.g. if a sentence is valid when starting with a noun followed by a verb:

```
sentence(C) :- noune(A), verb(B), append(A, B, C)
```

However this is a computation-heavy implementation -> not feasible.

We can however use difference lists:

```
X – X = []
[a, b, c] – [b] = [a, c]
...
```

DCG = Definite clause grammars
DCG is special notation for grammar representations, using difference lists:

```
sentence --> noun, verb.
// that's equivalent to the following:
sentence(A-C) :- noun(A-B), verb(B-C).
// you can call the first one with with sentence(YOUR_SENTENCE_AS_LIST, [])
```

12 Visual Programming

TODO