# MOV, Lambda, DSL

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM

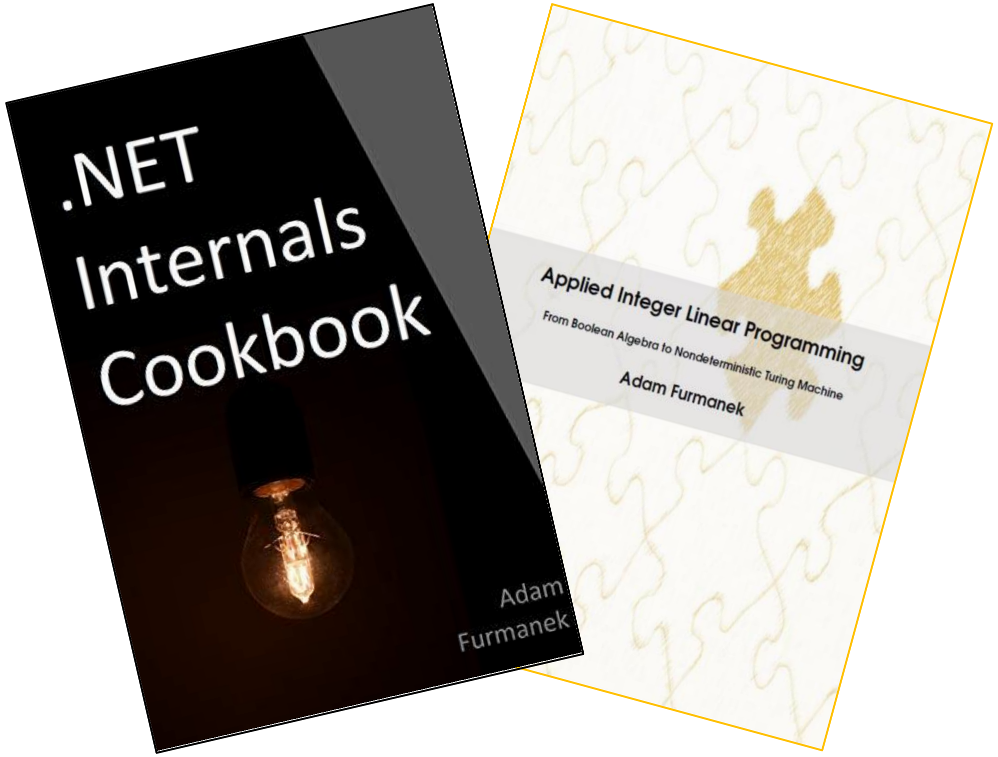
# About me

Software Engineer, Blogger, Book Writer, Public Speaker.
Author of ***Applied Integer Linear Programming*** and ***.NET Internals Cookbook***.

http://blog.adamfurmanek.pl

contact@adamfurmanek.pl

furmanekadam




Random IT Utensils

IT, operating systems, maths, and more.

# Agenda

What is abstraction.

From async down to MOV.

Turing Machine.

From function up to Typed Lambda Calculus.

Declarative forms and DSL

Summary.

# What is abstraction

# Abstraction according to Wikipedia

The process of **removing** physical, spatial, or temporal details or attributes in the study of objects or systems to **focus attention** on details of greater importance; it is similar in nature to the process of generalization.

The **creation** of abstract concept-objects by mirroring common features or attributes of various non-abstract objects or systems of study – the result of the process of abstraction.

# Abstraction = View

# Abstraction examples according to Wikipedia

The usage of **data types** to perform data abstraction to separate usage from working representations of data structures within programs.

The concept of **procedures, functions, or subroutines** which represent a specific of implementing control flow in programs.

The process of reorganizing common behavior from **non-abstract classes into "abstract classes"** using inheritance to abstract over sub-classes as seen in the object-oriented C++ and Java programming languages.

# What is our core entity? Computation

We want to be able to compute anything.

And we want our Computers to do that.

# From async down to MOV

LET'S SOLVE IT ALL!

# async programming

The goal is to unblock the thread.

Typically implemented as a coroutine transformation.

Implemented entirely in the compiler. Doesn’t require any support from the runtime.

```
var data = await SlowSumToN(10);
Console.WriteLine(data);
```

```
SlowSumToN(10).ContinueWith(data => Console.WriteLine(data)).Wait();
```

# async to state machine

```
public static async Task Main()
{
    var data = await SlowSumToN(10);
    Console.WriteLine(data);
}

static async Task<int> SlowSumToN(int N){
    await Task.Delay(1000);
    return Enumerable.Range(1, N).Sum();
}
```

```
class StateMachine {
    private int result;
    private Exception exception;
    private bool afterDelay = false;
    private int N;
    private TaskAwaiter delayAwaiter;

    void MoveNext(){
        try{
            if(!afterDelay) {
                delayAwaiter = Task.Delay(1000).GetAwaiter();
                this.afterDelay = true;
                ScheduleContinuationToRunMoveNextAgain();
                return;
            }

            this.result = Enumerable.Range(1, N).Sum();
        } catch (Exception e){
            this.exception = e;
        }
    }
}
```

# Enumeration to state machine

```
Enumerable.Range(1, N);
```

```
public static IEnumerable<int> Range(int start, int count) {
    long max = ((long)start) + count - 1;
    if (count < 0 || max > Int32.MaxValue) throw Error.ArgumentOutOfRange("count");
    return RangeIterator(start, count);
}

static IEnumerable<int> RangeIterator(int start, int count) {
    for (int i = 0; i < count; i++) yield return start + i;
}
```

# Enumeration to state machine

```
public static class Enumerable {
    public static IEnumerable<int> Range(int start, int count) {
        return new RangeEnumerable(start, count);
    }

    private class RangeEnumerable : IEnumerable<int> {
        private int _Start;
        private int _Count;

        public RangeEnumerable(int start, int count) {
            _Start = start;
            _Count = count;
        }

        public virtual IEnumerator<int> GetEnumerator() {
            return new RangeEnumerator(_Start, _Count);
        }

        IEnumerator IEnumerable.GetEnumerator() {
            return GetEnumerator();
        }
    }
}
```

```
private class RangeEnumerator : IEnumerator<int> {
    private int _Current;
    private int _End;

    public RangeEnumerator(int start, int count) {
        _Current = start - 1;
        _End = start + count;
    }

    public virtual void Dispose() {
        _Current = _End;
    }

    public virtual void Reset() {
        throw new NotImplementedException();
    }

    public virtual bool MoveNext() {
        ++_Current;
        return _Current < _End;
    }

    public virtual int Current { get { return _Current; } }
    object IEnumerator.Current { get { return Current; } }
}
```

# Enumeration to loop

```
Enumerable.Range(1, N).Sum()
```

```
var enumerator = Enumerable.Range(1, 10).GetEnumerator();

int sum = 0;
while(enumerator.MoveNext()){
    sum += enumerator.Current;
}

return sum;
```

# Loop to if + jump

```
int sum = 0;
while(enumerator.MoveNext()){
    sum += enumerator.Current;
}
```

```
int sum = 0;

loopHead:
    if(!enumerator.MoveNext()){
        jumpToEnd();
    }
    sum += enumerator.Current;
    jumpToLoopHead();

end:
    return sum;
```

# Story continues

This code is compiled to Intermediate Language (some .NET assembly-like language).

It is later JIT-compiled to a machine code.

Machine code has no idea about functions, variables, parameters.

All it knows is:

- Memory
- CPU registers
- Instruction Pointer (which instruction to execute)

Instructions:

- CMP – compare
- JMP – jump
- ADD, SUB, MUL, … - maths
- MOV – assign (move) data from one place to another

# MOV is powerful

MOV *to, from*

One mnemonic in an assembly language but 35 instructions on x86_64 architecture.

Can be used to modify CPU flags and registers.

Can be used to implement any other x86 instruction.

You need ONE assembly instruction to implement ANY application.

# M/o/Vfuscator2
[https://github.com/xoreaxeaxeax/movfuscator](https://github.com/xoreaxeaxeax/movfuscator)

```
<is_prime>:
  push  ebp
  mov   ebp,esp
  sub   esp,0x10
  cmp   DWORD PTR [ebp+0x8],0x1
  jne   8048490 <is_prime+0x13>
  mov   eax,0x0
  jmp   80484cf <is_prime+0x52>
  cmp   DWORD PTR [ebp+0x8],0x2
  jne   804849d <is_prime+0x20>
  mov   eax,0x1
  jmp   80484cf <is_prime+0x52>
  mov   DWORD PTR [ebp-0x4],0x2
  jmp   80484be <is_prime+0x41>
  mov   eax,DWORD PTR [ebp+0x8]
  cdq
  idiv  DWORD PTR [ebp-0x4]
  mov   eax,edx
  test  eax,eax
  jne   80484ba <is_prime+0x3d>
  mov   eax,0x0
  jmp   80484cf <is_prime+0x52>
  add   DWORD PTR [ebp-0x4],0x1
  mov   eax,DWORD PTR [ebp-0x4]
  imul  eax,DWORD PTR [ebp-0x4]
  cmp   eax,DWORD PTR [ebp+0x8]
  jle   80484a6 <is_prime+0x29>
  mov   eax,0x1
  leave
  ret
```

# OISC - One-Instruction Set Computer

MOV mnemonic on x86 represents multiple machine code instructions.

However, there are CPUs with literally one instruction which are still Turing Complete.

Typically:

- Subtract and branch if less than or equal to zero
- Subtract and branch if negative
- Subtract if positive else branch
- Reverse subtract and skip if borrow
- Subtract and branch if non zero (SBNZ a, b, c, destination)

# Sidenote
# ZISC – Zero-Instruction Set Computer

No instructions at all.

A very complex pattern matching.

Typically compared to neural networks.

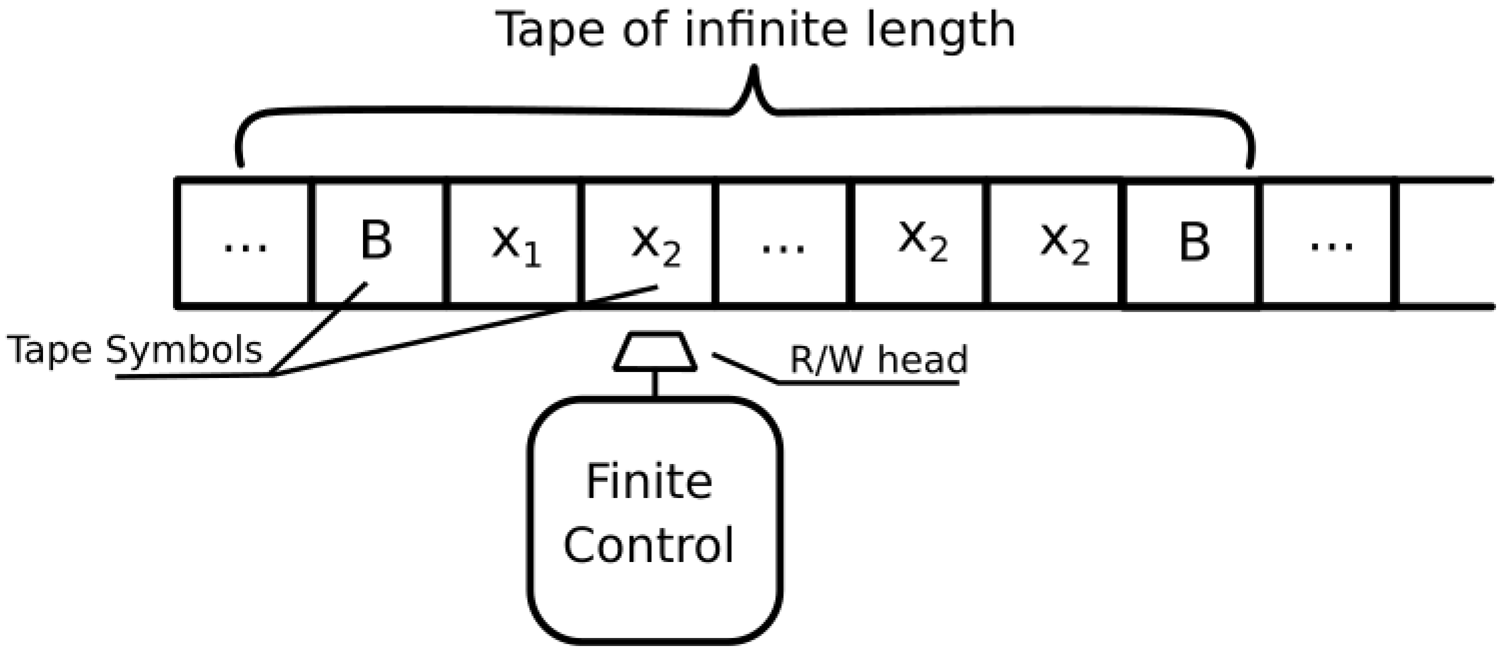
Used for image recognition.

# Turing Machine

# Turing Machine

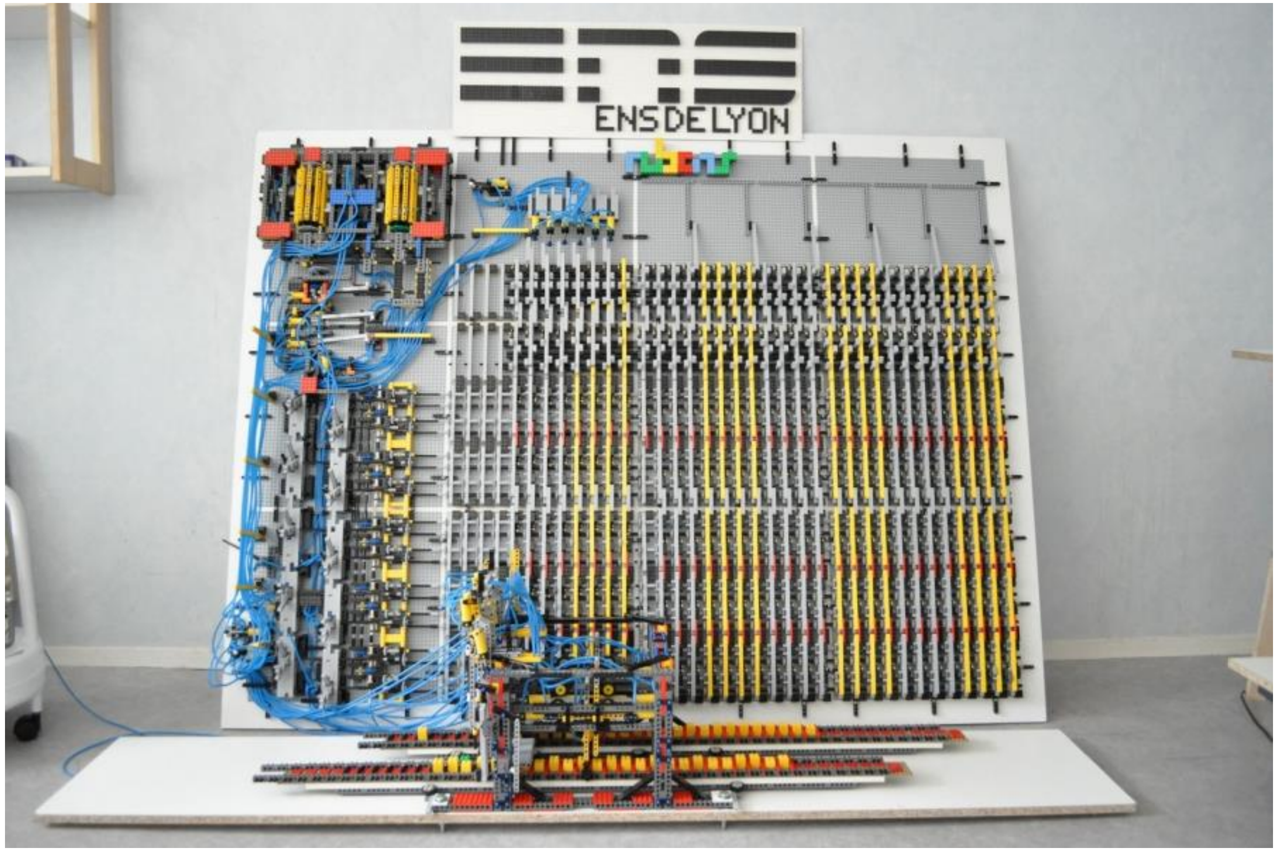
Mathematical model of computation.

It defines an abstract machine which manipulates symbols on an infinite tape according to a finite set of rules.




https://iq.opengenus.org/general-introduction-to-turing-machine/

# Turing Machine built with LEGO




https://upload.wikimedia.org/wikipedia/commons/7/7b/Lego_Turing_Machine.jpg

# Addition in Turing Machine

```
q0  - looking for the beginning, no carry
q1  - looking for the beginning, with carry
q2  - looking for digit of first number, no carry
q3  - looking for digit of first number, with carry
q4  - 0 from first number, looking for boundary between numbers
q5  - 1 from first number, looking for boundary between numbers
q6  - 2 from first number, looking for boundary between numbers
q7  - 0 from first number, looking for digit of second number
q8  - 1 from first number, looking for digit of second number
q9  - 2 from first number, looking for digit of second number
q10 - 0 in sum, looking for end of second number
q11 - 1 in sum, looking for end of second number
q12 - 2 in sum, looking for end of second number
q13 - 3 in sum, looking for end of second number
q14 - 0 in sum, looking for a place to write
q15 - 1 in sum, looking for a place to write
q16 - 2 in sum, looking for a place to write
q17 - 3 in sum, looking for a place to write
```

```
$aaaaaaa%bbbbbbbbb^.
```

```
$,q0,$,q2,R
%,q0,%,q0,L

^,q0,^,q0,L
0,q0,0,q0,L
1,q0,1,q0,L

$,q1,$,q3,R
%,q1,%,q1,L

^,q1,^,q1,L
0,q1,0,q1,L
1,q1,1,q1,L

0,q2,$,q4,R
1,q2,$,q5,R
%,q2,%,q7,R

0,q3,$,q5,R
1,q3,$,q6,R
%,q3,%,q8,R

0,q4,0,q4,R
1,q4,1,q4,R
%,q4,%,q7,R

0,q5,0,q5,R
1,q5,1,q5,R
%,q5,%,q8,R
```

```
0,q6,0,q6,R
1,q6,1,q6,R
%,q6,%,q9,R

%,q7,%,q7,R

0,q7,%,q10,R
1,q7,%,q11,R

%,q8,%,q8,R

0,q8,%,q11,R
1,q8,%,q12,R
^,q8,^,q15,R

%,q9,%,q9,R

0,q9,%,q12,R
1,q9,%,q13,R
^,q9,^,q16,R

0,q10,0,q10,R
1,q10,1,q10,R
^,q10,^,q14,R

0,q11,0,q11,R
1,q11,1,q11,R
^,q11,^,q15,R

0,q12,0,q12,R
1,q12,1,q12,R
^,q12,^,q16,R
```

```
0,q13,0,q13,R
1,q13,1,q13,R
^,q13,^,q17,R

0,q14,0,q14,R
1,q14,1,q14,R
.,q14,0,q0,L

0,q15,0,q15,R
1,q15,1,q15,R
.,q15,1,q0,L

0,q16,0,q16,R
1,q16,1,q16,R
.,q16,0,q1,L

0,q17,0,q17,R
1,q17,1,q17,R
.,q17,1,q1,L
```

# Algorithm complexity

Complexity of an algorithm is the amount of resources it requires to run.

Turing Machine is a simple model for comparing algorithms. Others include Random Access Machine, recursive functions or lambda calculus.

We count the numer of operations the algorithm needs to execute for a given input.

# Pros and Cons

Turing Machine can be used to calculate anything.

A programming language is Turing Complete if it's capable of calculating anything which can be calculated with a Turing Machine.

It's very simple to understand and implement.

Programming is hard.

It's nearly impossible to prove anything about the execution in an automated manner.

Can easily become incomprehensible when generated automatically.

Requires step-by-step thinking – we need to describe the algorithm precisely.

# From function up to Typed Lambda Calculus

LET'S MAKE IT PROVABLE!

# Lambda Calculus

Another model of computation.

Based on function abstraction and application.

Can simulate any Turing Machine.

3 rules (on the right).

In short: we have a function which accepts one parameter, contains any function as a body, and can call other functions.

No types, no constants, no literals.

Variable
- x

Abstraction
- $(\lambda x. M)$
- M is lambda term. x is bound in the expression
- Think of: *function(x) { M }*

Application
- $(M\ N)$
- Applying a function M to an argument N
- Think of: *M(N)* where M is function

# Beta reduction

Used to replace occurrences of a variable in a term with the variable.

$$(\lambda n.n \times 2)5 \ \rightarrow 5 \times 2$$

Basically a variable substitution.

# What can we do with that?

Literals:

$$TRUE = \lambda x. \lambda y. x$$

$$FALSE = \lambda x. \lambda y. y$$

Logical operators:

$$AND = \lambda p. \lambda q. p\ q\ p$$

$$OR = \lambda p. \lambda q. p\ p\ q$$

$$NOT = \lambda p. p\ FALSE\ TRUE$$

$$IFTHENELSE = \lambda p. \lambda a. \lambda b. p\ a\ b$$

Let's take $AND$ with first argument $TRUE$. We should return second argument as *true && x* reduces to *x*.

$\lambda p. \lambda q. p\ q\ p$ for $p = TRUE$

$\lambda q. TRUE\ q\ TRUE$ we substitute first TRUE

$\lambda q. (\lambda x. \lambda y. x\ q\ TRUE)$ we ignore second TRUE

$\lambda q. q$

Notice that the result is still a function. If we substitute $q$ with $TRUE$ then we end up with $TRUE$

$TRUE$ is „the literal" for truthy value. It's like *true* in other languages, even though it's a function.

# What can we do with that?

Numbers:
- $0 = \lambda f.\lambda x.x$ (notice that this is equal to FALSE).
- $1 = \lambda f.\lambda x.f\ x$
- $2 = \lambda f.\lambda x.f\ (f\ x)$
- $3 = \lambda f.\lambda x.f\ (f\ (f\ x))$

So number x is a function applied x times.
- $SUCC = \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$

But what are $f$ and $x$ here? They are functions.

We won't get to something „irreducible" or „without variables".

We'll get to some form which is equivalent to another one.

$ISZERO = \lambda n.n\ (\lambda x.FALSE)\ TRUE$

Let's calculate *ISZERO* for zero:

$\lambda n.n\ (\lambda y.FALSE)\ TRUE\ 0$ substitute 0

$\lambda f.\lambda x.x\ (\lambda y.FALSE)\ TRUE$ notice that f is ignored

$\lambda x.x\ TRUE$ reduce again

*ISZERO* for one:

$\lambda n.n\ (\lambda y.FALSE)\ TRUE\ 1$ *substitute 1*

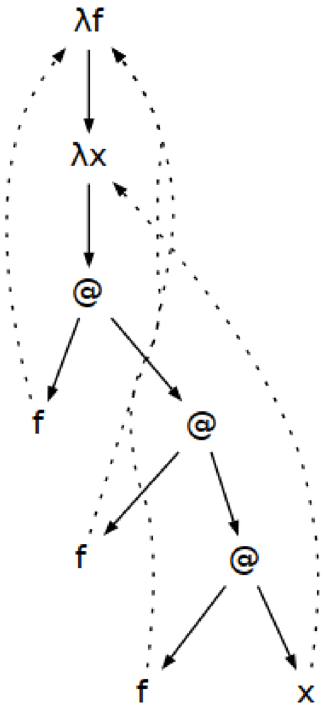
$\lambda f.\lambda x.(f\ x)\ (\lambda y.FALSE)\ TRUE$ *f is not ignored*

$\lambda x.((\lambda y.FALSE)\ x)\ TRUE$ *reduce again*

$(\lambda y.FALSE)\ TRUE$ *reduce again*

```
var three = 3;
```

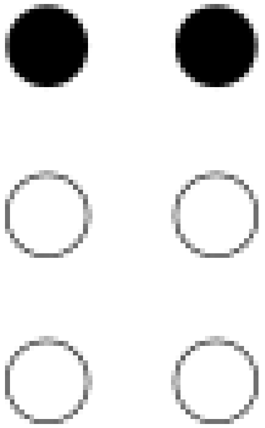
$$3 = \lambda f.\lambda x.f\ (f\ (f\ x))$$

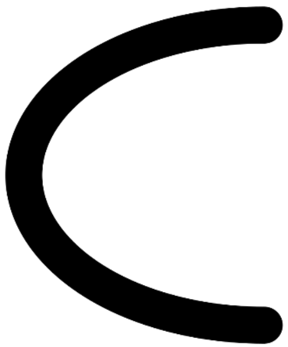



```
var three = 3;
```

$$3 = \lambda f.\lambda x.f\ (f\ (f\ x))$$

*III*





# Data structures

Structures are implemented in the same way.

We define some well-known literals and then provide accessors.

We start with basic ones like tuples. Then we move on to records, intersections and others.

By building more and more data structures we get Typed Lambda Calculus.

$PAIR = \lambda x.\lambda y.\lambda f.f\ x\ y$

$FIRST = \lambda p.p\ TRUE$

$SECOND = \lambda p.p\ FALSE$

$NIL = \lambda x.TRUE$

$NULL = \lambda p.p\ (\lambda x.\lambda y.FALSE)$

# Object-Oriented Programming

OOP can be implemented in Lambda Calculus in multiple ways.

One of the ideas:

- An object with fields is a tuple with components
- Each getter and seter is a selector of the tuple's component
- Inheritance is implemented as unions and intersections of objects

Similarly, we can translate Lambda Calculus to OOP.

At some point it's not about „what we can do" (because they become equivalent) but „how readable that is".

# Dependent types

These are types whose definitions depend on a value.

They are used to encode quantifiers (for all, exists).

They allow to verify that the application modifies state correctly.

```
data Vect : Nat -> Type -> Type where
  Nil  : Vect 0 a
  (::) : (x : a) -> (xs : Vect n a) -> Vect (n + 1) a

total
pairAdd : Num a => Vect n a -> Vect n a -> Vect n a
pairAdd Nil       Nil       = Nil
pairAdd (x :: xs) (y :: ys) = x + y :: pairAdd xs ys
```

# Combinatory Logic

We showed that it's possible to calculate anything with just one instruction in imperative programming.

A lambda in Lambda Calculus is allowed to have basically any body.

One might say that it's too much!

There are other systems apart from SKI and SK, for instance BCKW.

Identity combinator:

- $I = \lambda x.x$

Constant:

- $K = \lambda x.\lambda y.x$

Generalized application (apply x to y inside the environment z):

- $S = \lambda x.\lambda y.\lambda z.\left(x\ z\ (y\ z)\right)$

$I$ can be represented with composition of $S$ and $K$ so we need only two functions.

Number two:

$$2 = S(S(KS)K)I$$

# Pros and Cons

We can prove our programs work.

We can effectively compose concepts to build more complex abstractions.

It’s all based on maths so computers can reason about that.

You can’t „hack” things in the middle. You can’t cheat on math.

It requires a lot of discipline. Just like you may be tired of showing the compiler that your JSON is of some type – you’ll need to do way more to show that mathematically.

It cannot be executed directly on the CPU (even with SECD machine or Krivine machine).

# Church-Turing thesis

It states that a function on the natural numbers can be calculated by an effective method if and only if it is computable by a Turing Machine.

This means that whatever we can calculate with Turing Machine (= with imperative programming) can be calculated with lambda calculus.

It is also equivalent to general recursive functions – partial functions from natural numbers to natural numbers that are closed under basic operations.

What is an **effective method**?

It's something that a human can do with pen and paper:

- It consists of a finite numer of steps
- It always finishes
- It always produces a correct answer
- It is sufficient to follow rules rigorously (no need for ingenuity)

We do not have a formal, provable definitione of effective method. **Whole computer science is based on these 3 intuitive approaches**.

# Declarative forms and DSL

LET'S MAKE IT READABLE!

# Relational databases

We know data is there.

But we are not interested in how it’s accessed or how to calculate results – twelve laws of OLAP by Edgar Codd.

We just want to define what the result is and get it.

Alpha language:

http://www.inf.unibz.it/~franconi/teaching/2006/kbdb/Codd72a.pdf

QUEL:

```
range of E is EMPLOYEE
retrieve into W
(COMP = E.Salary / (E.Age - 18))
where E.Name = "Jones"
```

SEQUEL later named SQL (pronunced Es Kju El):

```
create table w as
select (e.salary / (e.age - 18)) as comp
from employee as e
where e.name = 'Jones'
```

# 3-satisfiability (3SAT)

We have a set of binary variables.

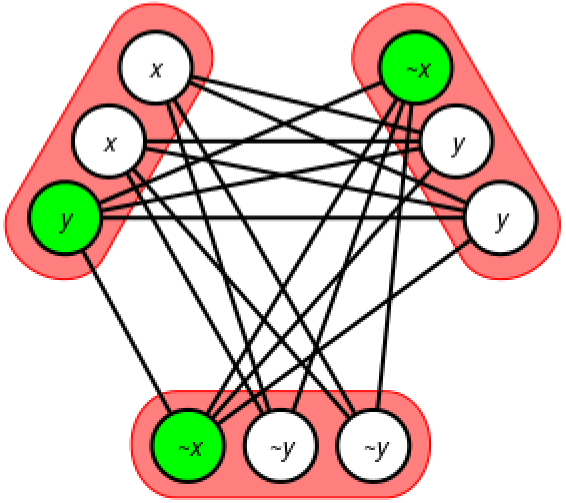
We have set of clauses – disjunctions of some variables.

$$x\ OR\ y\ OR\ {\sim}z$$

We want to find values for variables such that each clause is satisfied (returns true).

If clauses are limited to at most three literals then we get 3SAT.

One of the most important NP problems.



# Integer Linear Programming (ILP)

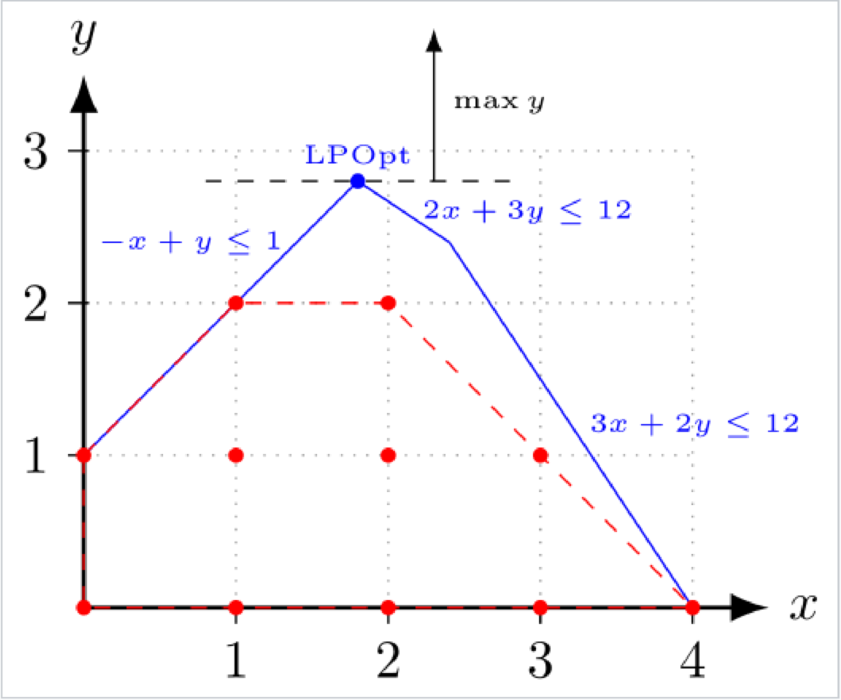
We have variables which can be real or integer.

We have set of linear formulas
- So we can add variables
- And we can multiply variables by constants

We can bound any formula with ≤ or ≥

We want to find a solution to the problem (and optimize some goal function).

$$\max y$$
$$-x + y \leq 1$$
$$3x + 2y \leq 12$$
$$2x + 3y \leq 12$$
$$x, y \geq 0$$
$$x, y \in \mathbb{Z}$$



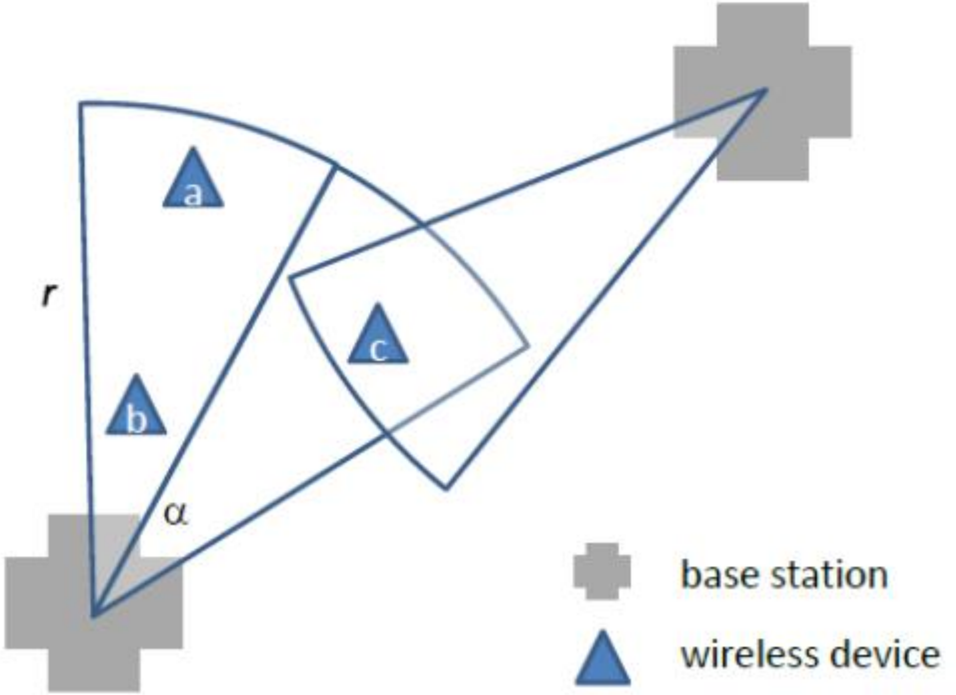

IP polytope with LP relaxation

# Why are they useful?

For some problems we can define the solution but we don't know how to calculate it.

We decouple problem definition from the problem calculation.

We can translate problems between models and use various solvers.




http://www.shivakasiviswanathan.com/MILCOM11.pdf

# Behavior-Driven Development tests

Allows for writing tests in a plaintext form.

Text is then translated into method invocations.

This improves collaboration between programmers and non-technical people.

Based on TDD and DDD, especially Ubiquitous language.

```
Scenario Outline: A user withdraws money from an ATM
    Given <Name> has a valid Credit or Debit card
    And their account balance is <OriginalBalance>
    When they insert their card
    And withdraw <WithdrawalAmount>
    Then the ATM returns <WithdrawalAmount>
    And their account balance is <NewBalance>

    Examples:
      | Name   | OriginalBalance | WithdrawalAmount | NewBalance |
      | Eric   | 100             | 45               | 55         |
      | Gaurav | 100             | 40               | 60         |
      | Ed     | 1000            | 200              | 800        |
```

# Domain-specific languages (DSL)

Much more expressive in their domain.

Less comprehensive for people from the outside.

Examples include HTML, scripting languages for game engines (like Unreal Engine), statistical modeling languages (R), Infrastructure-as-a-Code (IAAC) languages and more.

We can focus on the problem, not on the syntax!

# Summary

All these examples can be interchanged – nothing stops us from writing BDD tests in an assembly language.

The point is to use the right tool and to use it in the right way.

However, the problem is always the same. The only thing that changes is our perception.

We need to change the **view** and develop the right **abstraction**.

# Q&A

# References

Luca Cardelli, Martin Abadi - „A Theory of Objects (Monographs in Computer Science)”

Benjamin C. Pierce - „Types and Programming Languages”

Benjamin C. Pierce - „Advanced Topics in Types and Programming Languages”

J. Roger Hindley - „Lambda-Calculus and Combinators: An Introduction”

Adam Furmanek - „Applied Integer Linear Programming: From Boolean Algebra to Nondeterministic Turing Machine”

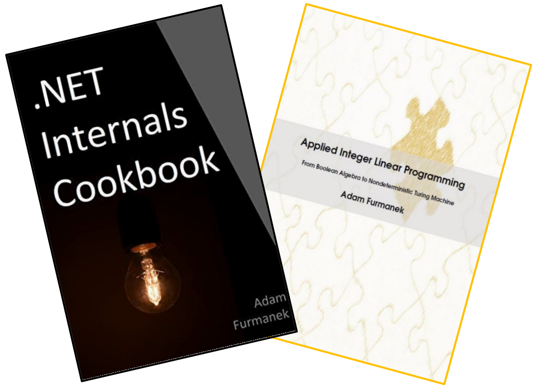
# References

https://crypto.stanford.edu/~blynn/lambda/ - Lambda Calculus

https://projectultimatum.org/cgi-bin/lambda - Lambda Calculus

https://komiamiko.me/math/ordinals/2020/06/21/ski-numerals.html - Combinatorial Logic

https://drops.dagstuhl.de/opus/volltexte/2021/14064/pdf/LIPIcs-ECOOP-2021-21.pdf - OOP in Lambda Calculus




Random IT Utensils

IT, operating systems, maths, and more.

# Thanks!

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM