Откуда что взялось в C#

Что повлияло на C#? На что повлиял C#?

Марк
Шевченко

mark-progmsk@yandex-team.ru

https://markshevchenko.pro

@markshevchenko

https://prog.msk.ru

Свойства

Свойства — C++ и Java?

C++

```
#include <iostream>

void main () {
  std::cout.width(10);
  std::cout << std::cout.width();
}

// =>         10
```

Java

```
import java.util.ArrayList;

public class Main {
  public static void main() {
    ArrayList<String> s =
      new ArrayList<String>();

    s.add(“foo”);
    s.add(“bar”);

    System.out.println(s.get(0));
    System.out.println(s.size());
  }
}
```

Свойства — C++ и Java?

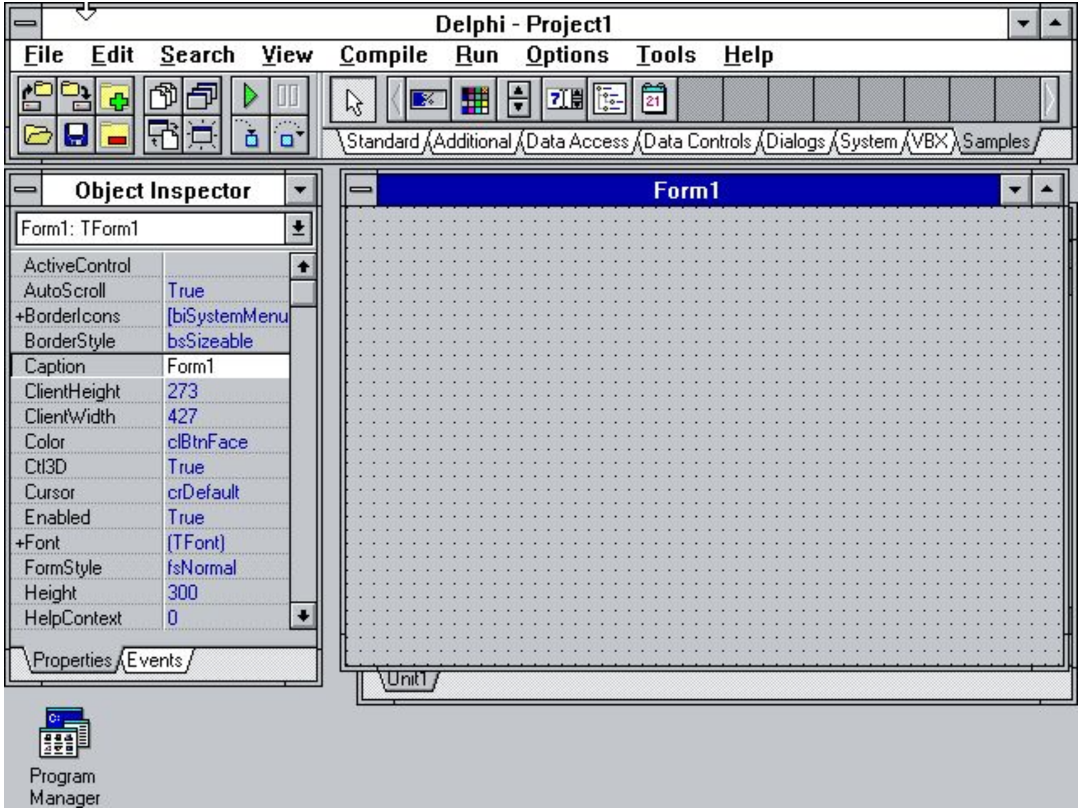
Java

```
public class Main {
  public static void main() {
    int[] a = new int[3];

    System.out.println(a.length);
  }
}
```

Java

```
import java.awt.geom.Arc2D;

public class Main {
  public static void main() {
    Arc2D shape =
      new Arc2D.Double(Arc2D.Pie);
    shape.setAngleExtent(200);

    System.out.println(
      shape.getAngleExtent()
    );
  }
}
```

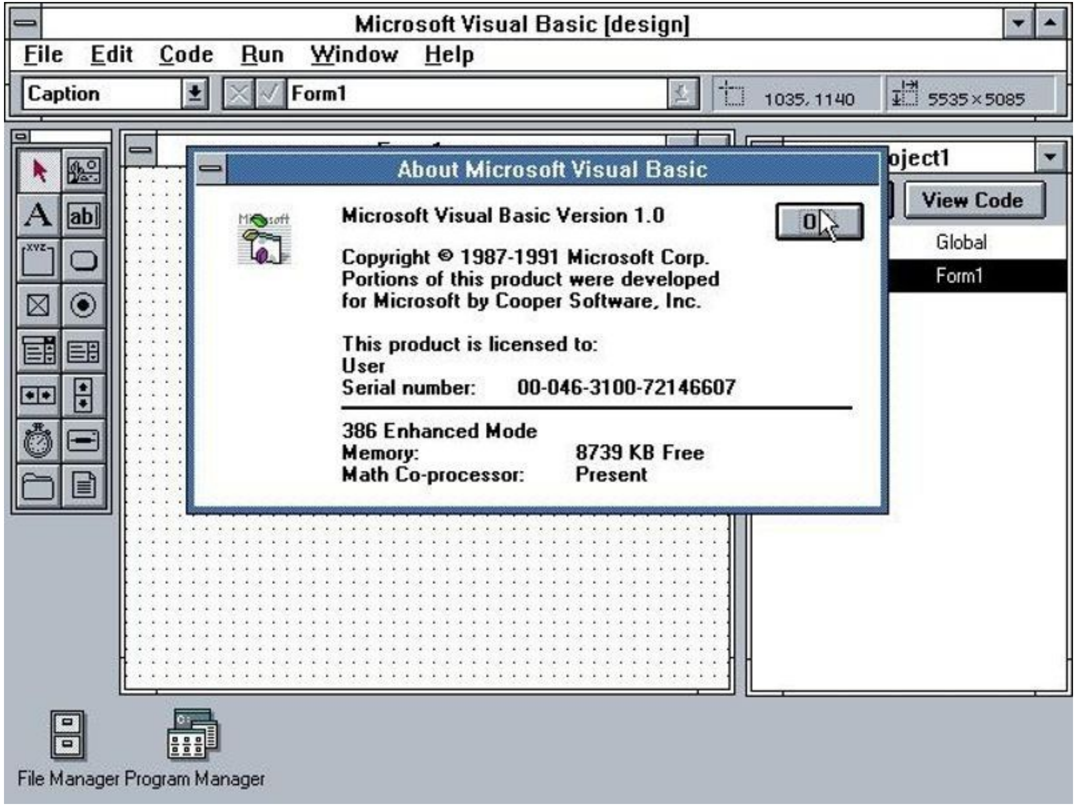
Свойства — Delphi (Object Pascal)



Form1: TForm1

ActiveControl	
AutoScroll	True
+BorderIcons	[biSystemMenu
BorderStyle	bsSizeable
Caption	Form1
ClientHeight	273
ClientWidth	427
Color	clBtnFace
Ctl3D	True
Cursor	crDefault
Enabled	True
+Font	(TFont)
FormStyle	fsNormal
Height	300
HelpContext	0

Свойства — Visual BASIC



About Microsoft Visual Basic

Microsoft Visual Basic Version 1.0

Copyright © 1987-1991 Microsoft Corp.
Portions of this product were developed
for Microsoft by Cooper Software, Inc.

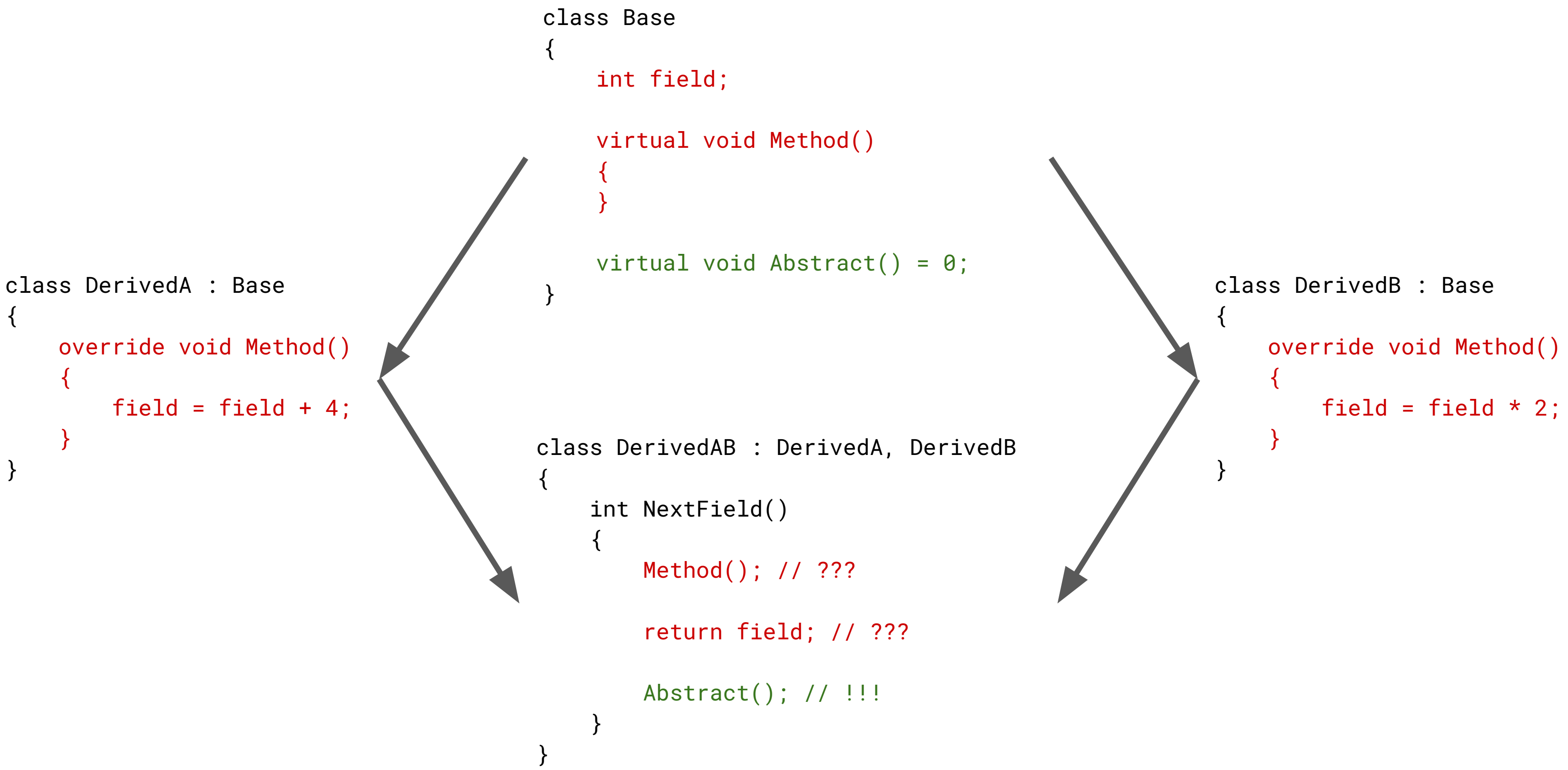
This product is licensed to:
User
Serial number: 00-046-3100-72146607

386 Enhanced Mode
Memory: 8739 KB Free
Math Co-processor: Present

Свойства — синтаксический сахар, но…

```
interface IFieldAndProperty
{
    int field;

    int Property { get; set; }
}
```

Свойства — синтаксический сахар, но…

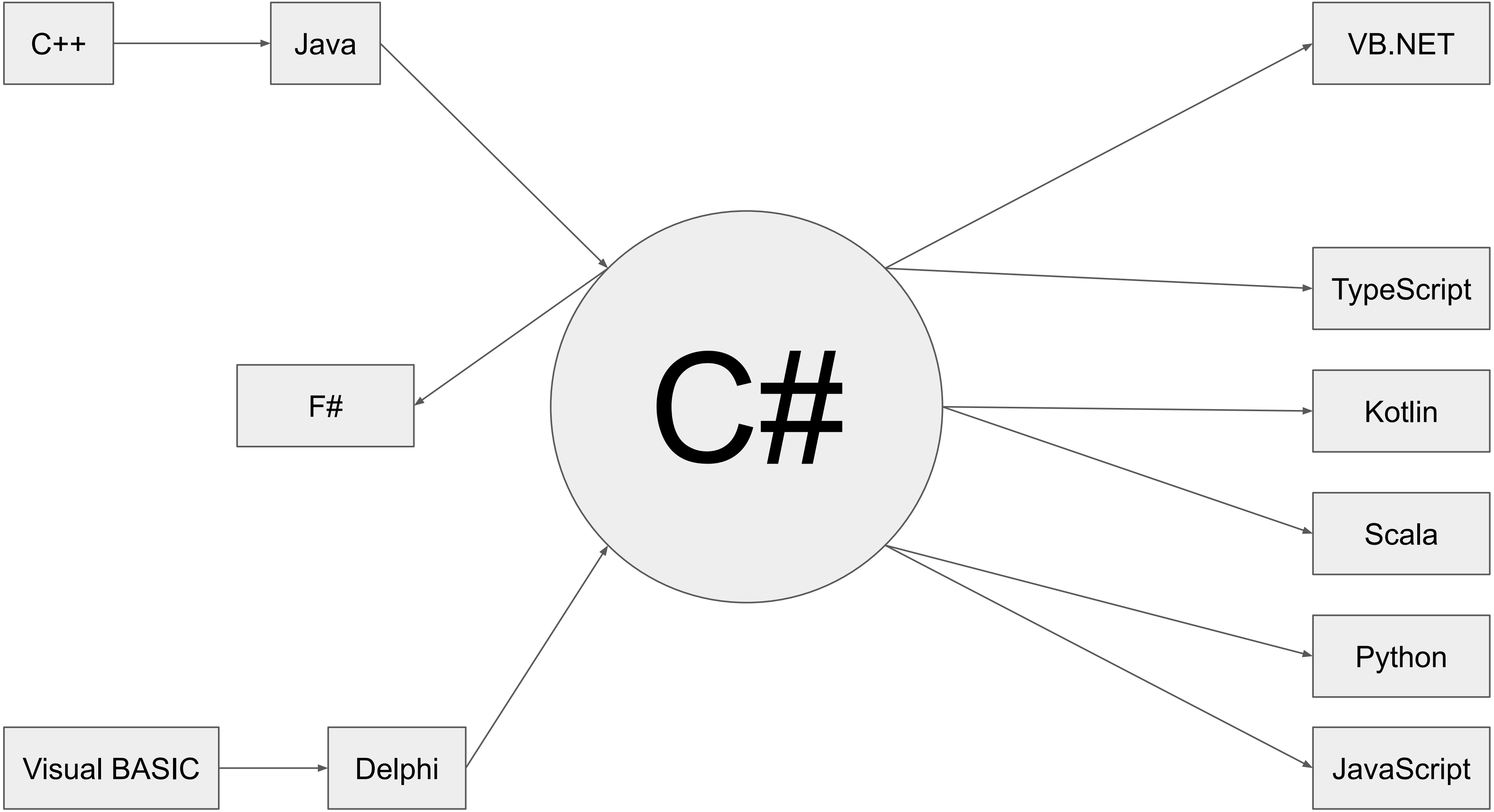


```
class Base
{
    int field;

    virtual void Method()
    {
    }

    virtual void Abstract() = 0;
}
```

```
class DerivedA : Base
{
    override void Method()
    {
        field = field + 4;
    }
}
```

```
class DerivedB : Base
{
    override void Method()
    {
        field = field * 2;
    }
}
```

```
class DerivedAB : DerivedA, DerivedB
{
    int NextField()
    {
        Method(); // ???

        return field; // ???

        Abstract(); // !!!
    }
}
```


C++
Java
VB.NET
TypeScript
F#
C#
Kotlin
Scala
Python
Visual BASIC
Delphi
JavaScript

События

```
interface IFieldPropertyDelegateAndEvent
{
    int field;

    int Property { get; set; }

    Action<object, EventArgs> action;

    event Action<object, EventArgs> Event;
}
```

Отличие событий от делегатов?

```
> Func<int, int> f = i => i + 2;
> f(2)
4
> f += i => i * 4;
> f(2)
8
```

Делегаты — как это было в C

```
#include <stdio.h>
#include <stdlib.h>

int int_compare(const int* a, const int* b) {
  return *a - *b;
}

void main() {
  int a[7] = { 23, 17, 34, 75, 45, 94, 21 };
  qsort(a, 7, sizeof(int), int_compare);

  for (size_t i = 0; i < 7; i++)
    printf("%d\n", a[i]);
}
```

Делегаты — как это стало в C++

```
class Foo
{
private:
  int value;

public:
  Foo(int value) { this->value = value; }

  int get_value() const { return value; }

  int compare(const Foo& other) const {
    return value - other.value;
  }

  static int compare2(const void* a, const void* b) {
    return static_cast<const Foo*>(a)->compare(*static_cast<const Foo*>(b));
  }
};
```

Делегаты — как это стало в Java

```
import java.util.Arrays;
import java.util.Comparator;

class Main {
  public static void main(String[] args) {
    Integer[] a = new Integer[] { 23, 17, 34, 75, 45, 94, 21 };

    Arrays.sort(a, new Comparator<Integer>() {
      public int compare(Integer a, Integer b) {
        return a - b;
      }
    });

    for (int i = 0; i < a.length; i++)
      System.out.println(a[i]);
  }
}
```

Делегаты — как это стало в C#

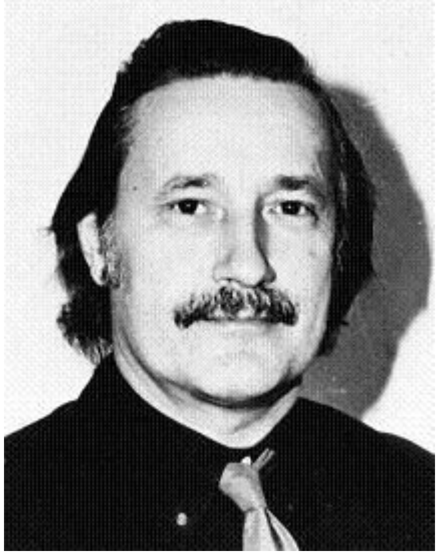
```
using System;

class Program {
  static int Compare(int a, int b) {
    return a - b;
  }

  public static void Main (string[] args) {
    int[] a = new int[] { 23, 17, 34, 75, 45, 94, 21 };
    Array.Sort(a, Compare);

    for (int i = 0; i < a.Length; i++)
      Console.WriteLine(a[i]);
  }
}
```

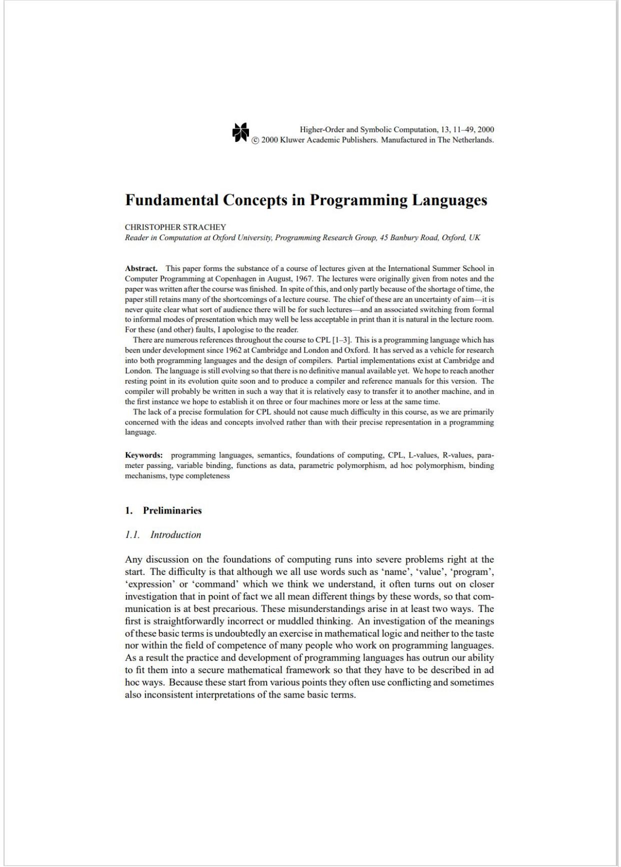
Делегаты сейчас

```
> delegate int F(int value);
> F f = i => i + 2;
> f(1)
3
> f += i => i * 2;
> f(1)
2

> Func<int, int> f = i => i + 2;
> f(1)
3
> f += i => i * 4;
> f(1)
2
```

Полиморфизм

Полиморфизм





Higher-Order and Symbolic Computation, 13, 11–49, 2000

Fundamental Concepts in Programming Languages

CHRISTOPHER STRACHEY
Reader in Computation at Oxford University, Programming Research Group, 45 Banbury Road, Oxford, UK

Abstract. This paper forms the substance of a course of lectures given at the International Summer School in Computer Programming at Copenhagen in August, 1967. The lectures were originally given from notes and the paper was written after the course was finished. In spite of this, and only partly because of the shortage of time, the paper still retains many of the shortcomings of a lecture course. The chief of these are an uncertainty of aim—it is never quite clear what sort of audience there will be for such lectures—and an associated switching from formal to informal modes of presentation which may well be less acceptable in print than it is natural in the lecture room. For these (and other) faults, I apologise to the reader.

There are numerous references throughout the course to CPL [1–3]. This is a programming language which has been under development since 1962 at Cambridge and London and Oxford. It has served as a vehicle for research into both programming languages and the design of compilers. Partial implementations exist at Cambridge and London. The language is still evolving so that there is no definitive manual available yet. We hope to reach another resting point in its evolution quite soon and to produce a compiler and reference manuals for this version. The compiler will probably be written in such a way that it is relatively easy to transfer it to another machine, and in the first instance we hope to establish it on three or four machines more or less at the same time.

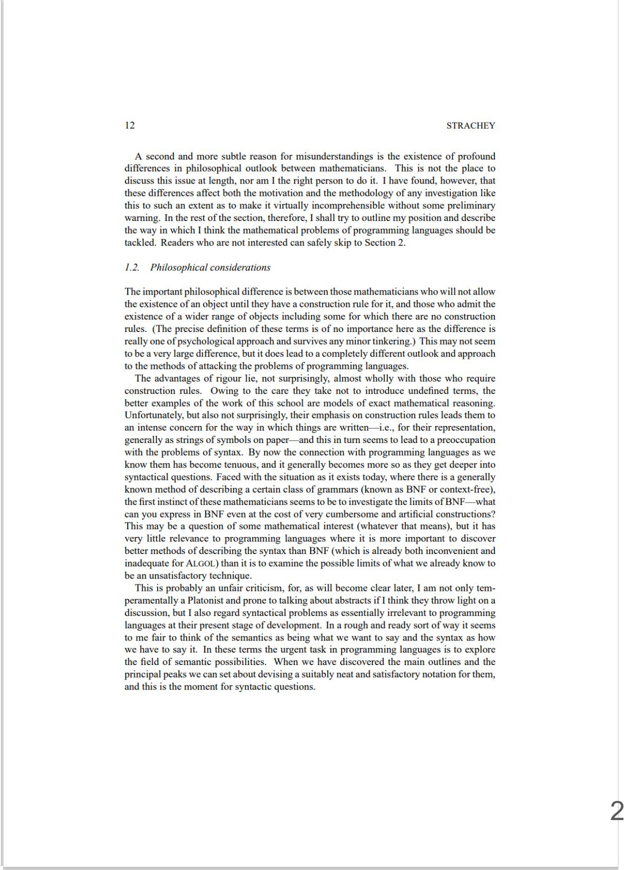
The lack of a precise formulation for CPL should not cause much difficulty in this course, as we are primarily concerned with the ideas and concepts involved rather than with their precise representation in a programming language.

Keywords: programming languages, semantics, foundations of computing, CPL, L-values, R-values, parameter passing, variable binding, functions as data, parametric polymorphism, ad hoc polymorphism, binding mechanisms, type completeness

1. Preliminaries

1.1. Introduction

Any discussion on the foundations of computing runs into severe problems right at the start. The difficulty is that although we all use words such as ‘name’, ‘value’, ‘program’, ‘expression’ or ‘command’ which we think we understand, it often turns out on closer investigation that in point of fact we all mean different things by these words, so that communication is at best precarious. These misunderstandings arise in at least two ways. The first is straightforwardly incorrect or muddled thinking. An investigation of the meanings of these basic terms is undoubtedly an exercise in mathematical logic and neither to the taste nor within the field of competence of many people who work on programming languages. As a result the practice and development of programming languages has outrun our ability to fit them into a secure mathematical framework so that they have to be described in ad hoc ways. Because these start from various points they often use conflicting and sometimes also inconsistent interpretations of the same basic terms.



A second and more subtle reason for misunderstandings is the existence of profound differences in philosophical outlook between mathematicians. This is not the place to discuss this issue at length, nor am I the right person to do it. I have found, however, that these differences affect both the motivation and the methodology of any investigation like this to such an extent as to make it virtually incomprehensible without some preliminary warning. In the rest of the section, therefore, I shall try to outline my position and describe the way in which I think the mathematical problems of programming languages should be tackled. Readers who are not interested can safely skip to Section 2.

1.2. Philosophical considerations

The important philosophical difference is between those mathematicians who will not allow the existence of an object until they have a construction rule for it, and those who admit the existence of a wider range of objects including some for which there are no construction rules. (The precise definition of these terms is of no importance here as the difference is really one of psychological approach and survives any minor tinkering.) This may not seem to be a very large difference, but it does lead to a completely different outlook and approach to the methods of attacking the problems of programming languages.

The advantages of rigour lie, not surprisingly, almost wholly with those who require construction rules. Owing to the care they take not to introduce undefined terms, the better examples of the work of this school are models of exact mathematical reasoning. Unfortunately, but also not surprisingly, their emphasis on construction rules leads them to an intense concern for the way in which things are written—i.e., for their representation, generally as strings of symbols on paper—and this in turn seems to lead to a preoccupation with the problems of syntax. By now the connection with programming languages as we know them has become tenuous, and it generally becomes more so as they get deeper into syntactical questions. Faced with the situation as it exists today, where there is a generally known method of describing a certain class of grammars (known as BNF or context-free), the first instinct of these mathematicians seems to be to investigate the limits of BNF—what can you express in BNF even at the cost of very cumbersome and artificial constructions? This may be a question of some mathematical interest (whatever that means), but it has very little relevance to programming languages where it is more important to discover better methods of describing the syntax than BNF (which is already both inconvenient and inadequate for ALGOL) than it is to examine the possible limits of what we already know to be an unsatisfactory technique.

This is probably an unfair criticism, for, as will become clear later, I am not only temperamentally a Platonist and prone to talking about abstracts if I think they throw light on a discussion, but I also regard syntactical problems as essentially irrelevant to programming languages at their present stage of development. In a rough and ready sort of way it seems to me fair to think of the semantics as being what we want to say and the syntax as how we have to say it. In these terms the urgent task in programming languages is to explore the field of semantic possibilities. When we have discovered the main outlines and the principal peaks we can set about devising a suitably neat and satisfactory notation for them, and this is the moment for syntactic questions.

Ad hoc полиморфизм

```
decimal Sum(this IEnumerable<decimal> source) {…}

double Sum(this IEnumerable<double> source) {…}

int Sum(this IEnumerable<int> source) {…}

long Sum(this IEnumerable<long> source) {…}

decimal? Sum(this IEnumerable<decimal?> source) {…}

double? Sum(this IEnumerable<double?> source) {…}

int? Sum(this IEnumerable<int?> source) {…}

long? Sum(this IEnumerable<long?> source) {…}
```

Generic Math в C# 11

```
public static TResult Sum<T, TResult>(IEnumerable<T> values)
    where T : INumber<T>
    where TResult : INumber<TResult>
{
    TResult result = TResult.Zero;

    foreach (var value in values)
    {
        result += TResult.Create(value);
    }

    return result;
}
```

Параметрический полиморфизм в C++

```
#include <iostream>

template <int N> struct Factorial
{
  enum { value = N * Factorial<N - 1>::value };
};

template <> struct Factorial<0>
{
  enum { value = 1 };
};

void main()
{
  std::cout << 4 << "!" << " = " << Factorial<4>::value << "\n";
  std::cout << 8 << "!" << " = " << Factorial<8>::value << "\n";
  std::cout << 16 << "!" << " = " << Factorial<16>::value << "\n";
}
```

Параметрический полиморфизм в Java

```
class DoublePoint { public double x; public double y; }

String json = """
  {
    "foo": { "x": 1.0, "y": 2 },
    "bar": { "x": 3, "y": 4.0 },
    "baz": { "x": 5, "y": 6 }
  }
""";

Map<String, DoublePoint> m = new Gson().fromJson(json, Map.class);

for (Map.Entry<String, DoublePoint> p: m.entrySet()) {
  System.out.println(p.getKey());
  System.out.println(p.getValue().x);
  System.out.println(p.getValue().y);
}
```

Параметрический полиморфизм в C#

```
public class Node<T>
{
    public T Value { get; set; }

    public Node<T> Next { get; set; }
}

public class List<T>
{
    public Node<T> Head { get; set; }
}
```

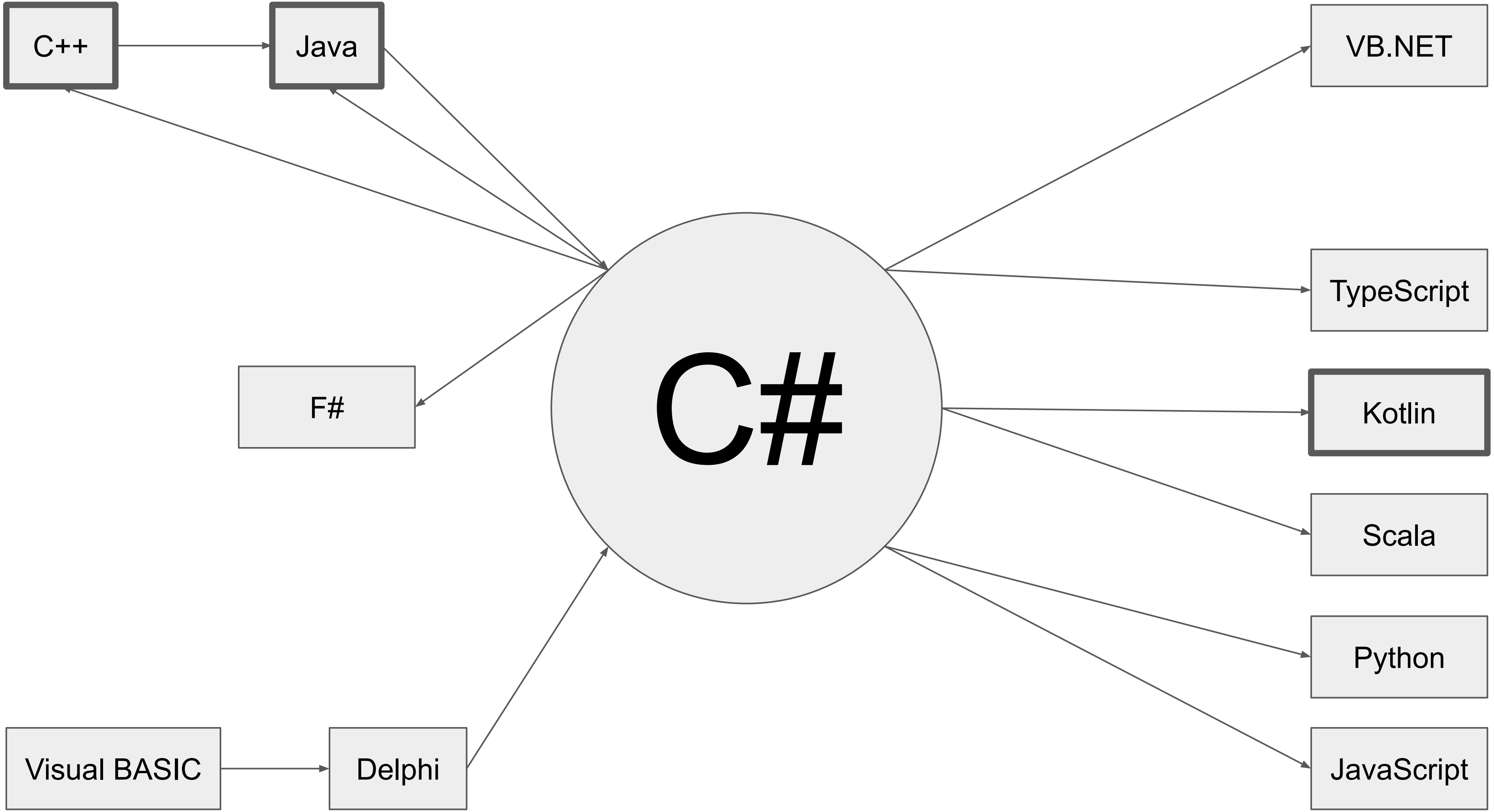
```
public bool Contains(T value)
{
    var current = this.Head;

    while (current != null)
    {
        if (current.value == value)
            return true;

        current = current.next;
    }

    return false;
}
```

Вывод типов

```
type Node<'a> = {
    value: 'a
    next: Node<'a> option
}

type List<'a> = {
    head: Node<'a> option
}
```

```
let contains value list =
  let rec check node =
      match node with
      | None -> false
      | Some node ->
        if node.value = value
        then true
        else check node.next

  check list.head
```

Вывод типов


C++
Java
VB.NET
TypeScript
F#
C#
Kotlin
Scala
Python
Visual BASIC
Delphi
JavaScript

LINQ

```
DateTime[] dateTimes = …;

// Понедельники в хронологическом порядке
var result = dateTimes.Where(x => x.DayOfWeek == DayOfWeek.Monday)
                      .OrderBy(x => x);
```

Почему лямбда-функции?

$$x^2 + y^2 = R^2$$

Почему лямбда-функции?

$$x^2 + y^2 = R^2$$

$$f(x, y) = \sqrt{x^2 + y^2}$$

$$f(3, 4) = 5$$

Почему лямбда-функции?

$$\sqrt{x^2 + y^2}(3, 4)$$

Почему лямбда-функции?

$$\sqrt{x^2 + y^2}(3, 4)$$

$$\widehat{xy}\sqrt{x^2 + y^2}$$

Почему лямбда-функции?

$$\sqrt{x^2 + y^2}(3, 4)$$

$$\widehat{xy}\sqrt{x^2 + y^2}$$

$$\wedge xy\sqrt{x^2 + y^2}$$

Почему лямбда-функции?

$$\lambda xy\sqrt{x^2 + y^2}$$

Почему лямбда-функции?

$$\lambda xy\sqrt{x^2 + y^2}$$

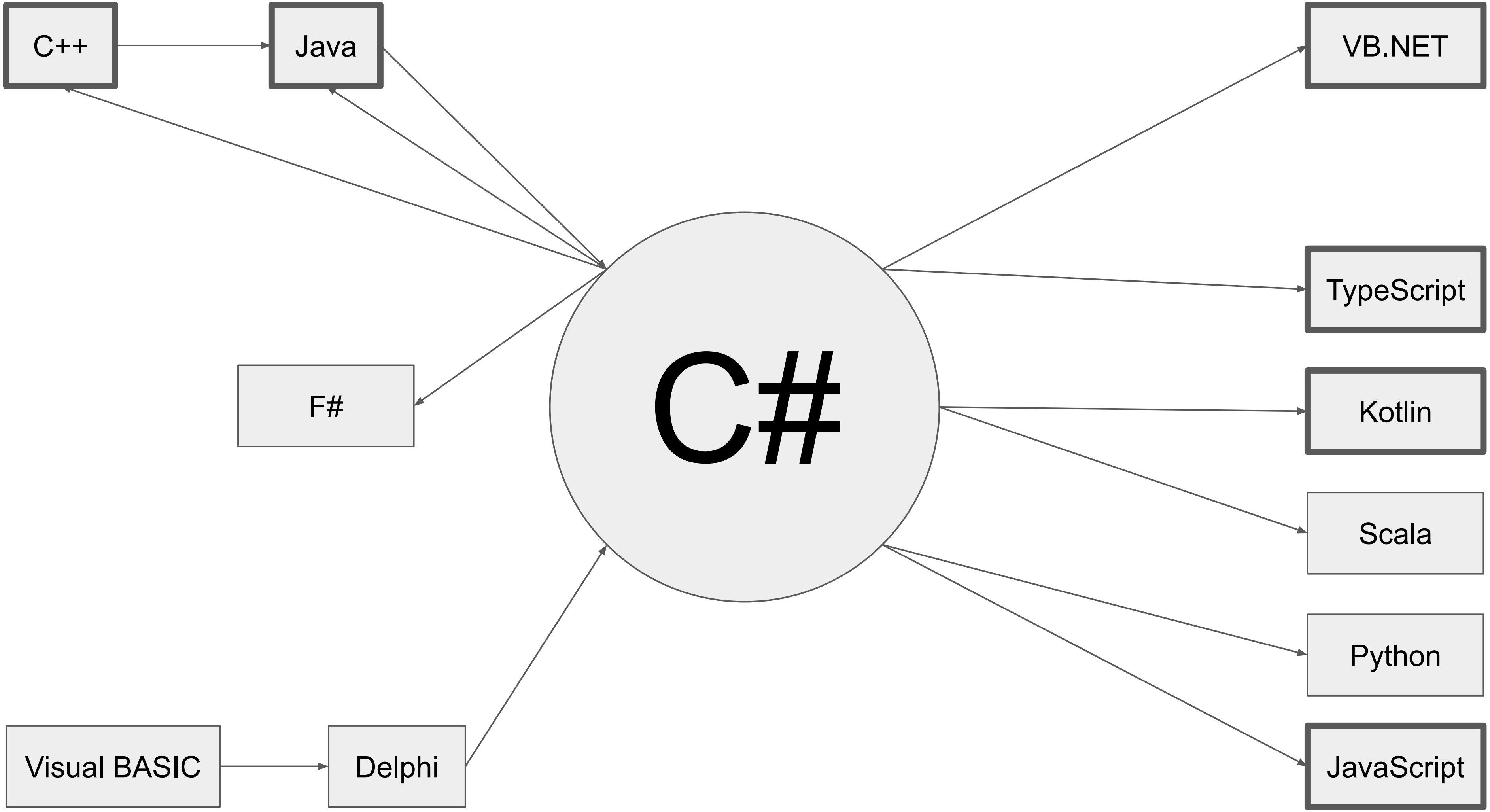
```
(lambda (x y) (sqrt (+ (* x x) (* y y))))
```

Почему лямбда-функции?

$$\lambda xy\sqrt{x^2 + y^2}$$

```
(lambda (x y) (sqrt (+ (* x x) (* y y))))
```

```
(x, y) => Math.Sqrt(x * x + y * y)
```


C++
Java
VB.NET
TypeScript
F#
C#
Kotlin
Scala
Python
Visual BASIC
Delphi
JavaScript

Деревья выражений

Цитирование в LISP

```
> (+ (/ 1 1) (/ 1 1) (/ 1 2) (/ 1 6) (/ 1 24) (/ 1 120) (/ 1 720) (/ 1 5040))
2.7182539682539684

> ’(+ (/ 1 1) (/ 1 1) (/ 1 2) (/ 1 6) (/ 1 24) (/ 1 120) (/ 1 720) (/ 1 5040))
(+ (/ 1 1) (/ 1 1) (/ 1 2) (/ 1 6) (/ 1 24) (/ 1 120) (/ 1 720) (/ 1 5040))
```

Макросы

```
(define (display-value value)
    (display value)
    (display " = ")
    (display (eval value))
    (newline))

(define a 15)
(define x 3)

(display-value ‘(* a x))
(* a x) = 45
```

Цитирование в F#

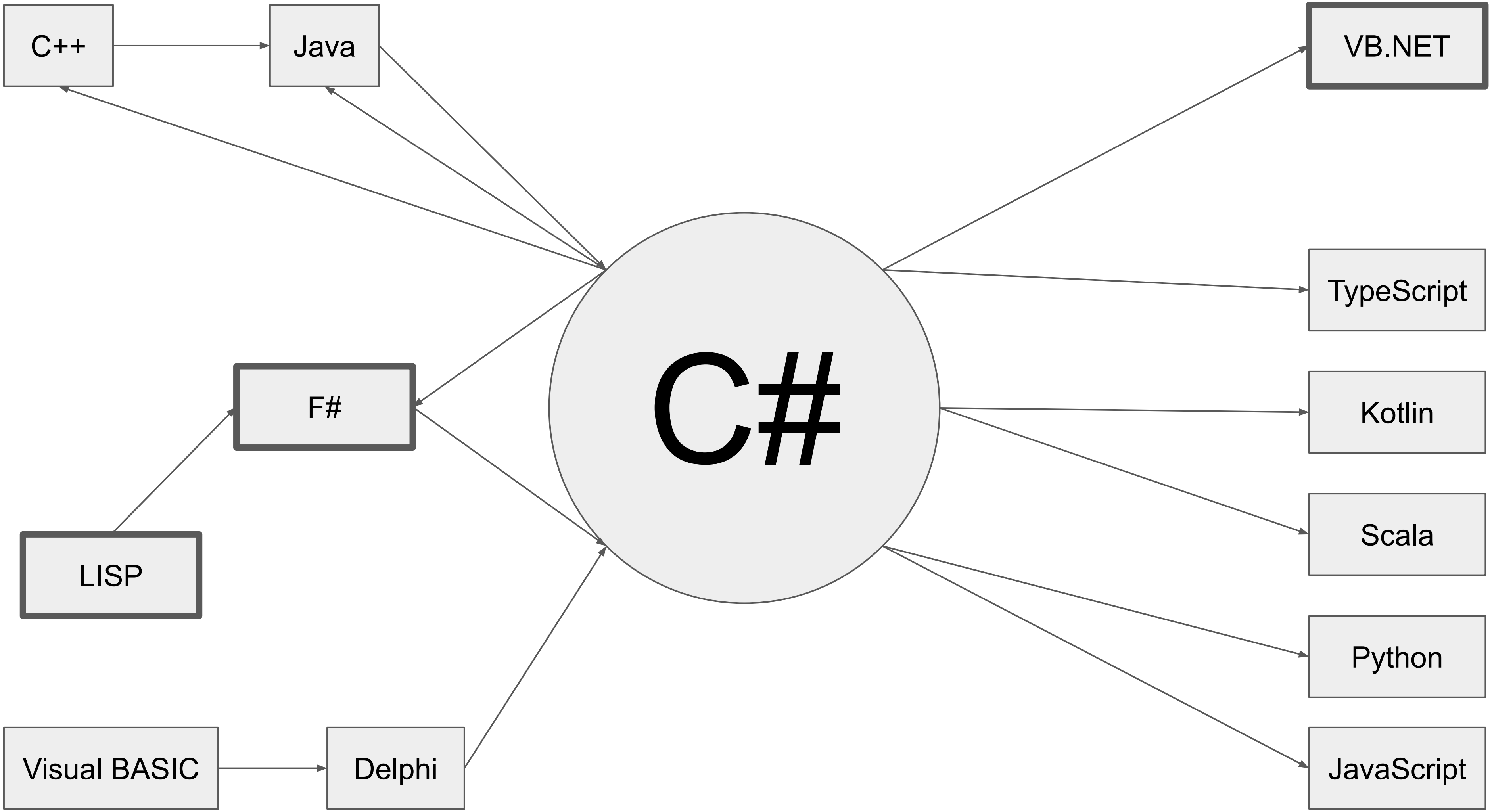
```
> let f = <@ fun x -> x + 2 @>

val f: Quotations.Expr<(int -> int)> =
  Lambda (x, Call (None, op_Addition, [x, Value (2)]))
```

Деревья выражений

```
> Func<double, double> square = x => x * x;
> square(2)
4

> Expression<Func<double, double>> expSquare = x => x * x;
> expSquare(2)
error CS1955: Невызываемый член "expSquare" не может использоваться как метод.

> expSquare.Compile()(2)
4
```

Производная

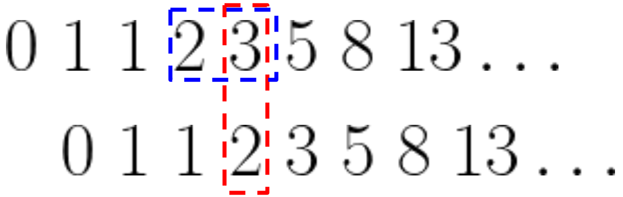
```
> #r "SySharp.dll"
> using SySharp;
> Symbolic.Derivative(x => x + 3).ToString()
"x => (1 + 0)"

> Symbolic.Derivative(x => x + 3).Simplify().ToString()
"x => 1"

> var a = 15;
> Symbolic.Derivative(x => a * x * (x + 3)).Simplify().ToString()
"x => (a * x) + (a * (x + 3))"
```


C++
Java
VB.NET
TypeScript
F#
C#
Kotlin
LISP
Scala
Python
Visual BASIC
Delphi
JavaScript

Ленивые вычисления

Haskell

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```



$0\ 1\ 1\ 2\ 3\ 5\ 8\ 13\ldots$

$0\ 1\ 1\ 2\ 3\ 5\ 8\ 13\ldots$

Ленивые вычисления — целые числа

```
static IEnumerable<BigInteger> Integers()
{
    var i = BigInteger.One;

    while (true)
        yield return i++;
}

// Integers(): 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,…
```

Ленивые вычисления — простые числа

```
static IEnumerable<BigInteger> Primes()
{
    return Integers().Where(IsPrime);
}

// Primes(): 2, 3, 5, 7, 11, 13, 17, 19, 23, 29,…
```

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 . . .

2 ⊕ 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 . . .

2 3 ⊕ 5 7 ~~9~~ 11 13 ~~15~~ 17 19 ~~21~~ 23 25 . . .

2 3 5 ⊕ 7 11 13 17 19 23 ~~25~~ 29 31 . . .

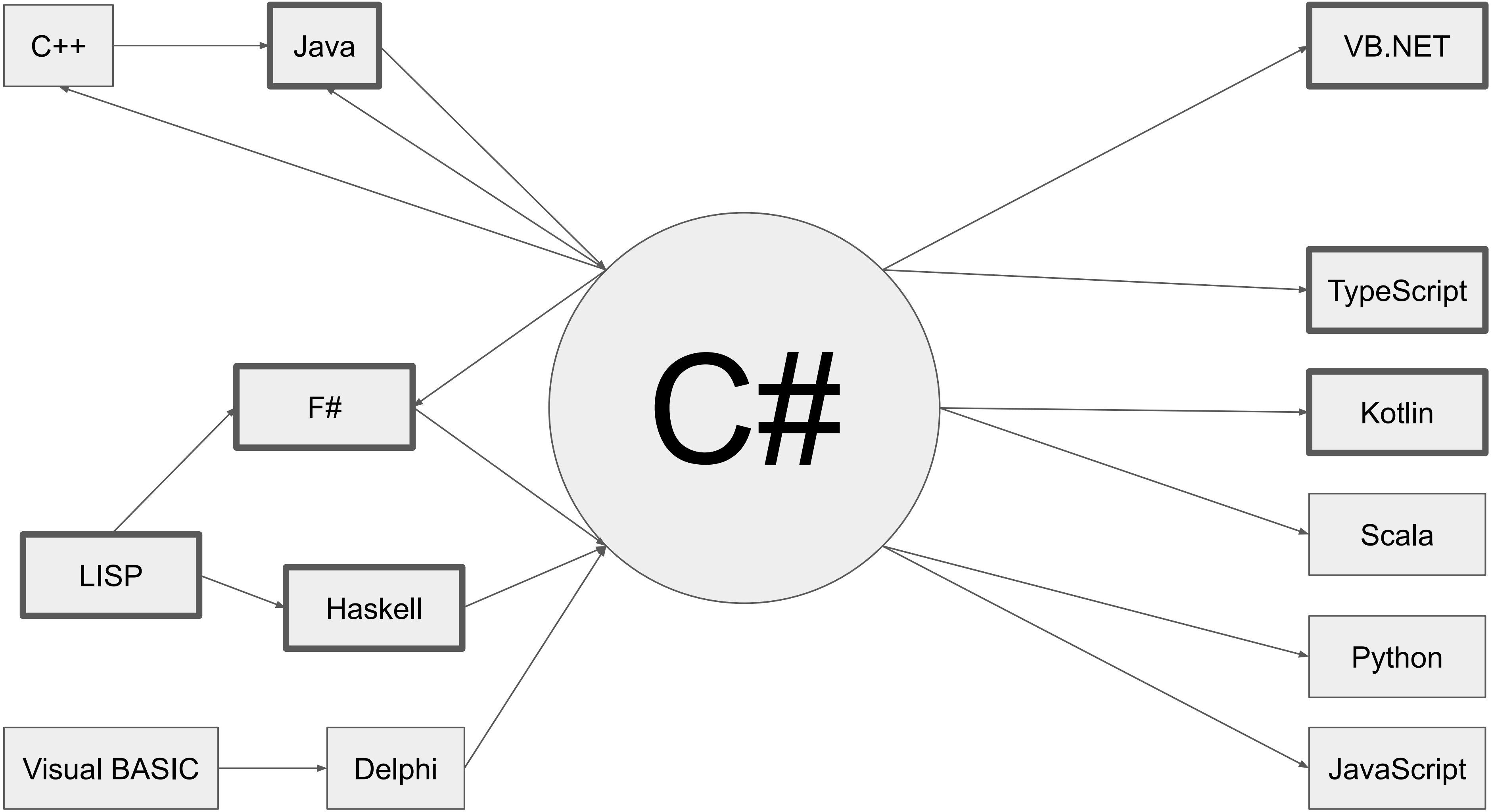
Ленивые вычисления — другие простые числа

```
static IEnumerable<BigInteger> Primes()
{
    return Integers().Skip(1).PrimesRecursive();
}

static IEnumerable<BigInteger> PrimesRecursive(this IEnumerable<BigInteger> s)
{
    var (nextPrime, tailPrimes) = s.HeadTail();
    yield return nextPrime;

    var filteredTail = tailPrimes.Where(i => i % nextPrime != BigInteger.Zero)
                                 .PrimesRecursive();

    foreach (var prime in filteredTail)
        yield return prime;
}

// Primes(): 2, 3, 5, 7, 11, 13, 17, 19, 23, 29,…
```

```
static IEnumerable<BigInteger> Fibs()
{
    yield return BigInteger.Zero;
    yield return BigInteger.One;

    foreach (var s in Fibs().Zip(Fibs().Skip(1), (a, b) => a + b))
        yield return s;
}

// Fibs(): 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,…
```

Ленивые вычисления — практический пример

```
public static IEnumerable<RegistryKey> GetAllKeys(RegistryKey root)
{
    yield return root;

    foreach (var subKeyName in root.GetSubKeyNames())
    {
        using (var subKey = root.OpenSubKey(subKeyName))
        {
            foreach (var descendantKey in GetAllKeys(subKey))
                yield return descendantKey;
        }
    }
}
```


C++
Java
VB.NET
TypeScript
C#
F#
Kotlin
LISP
Haskell
Scala
Python
Visual BASIC
Delphi
JavaScript

Асинхронность

Callback Hell

```
static void AsyncProcessRequest(IAsyncResult asyncResult)
{
    var listener = (HttpListener)asyncResult.AsyncState;
    listener!.BeginGetContext(AsyncProcessRequest, listener);
    var context = listener.EndGetContext(asyncResult);

    if (context.Request.HttpMethod == "GET")
    {
        if (context.Request.RawUrl == "/")
        {
            context.Response.StatusCode = 200;
            var buffer = Encoding.UTF8.GetBytes("<!DOCTYPE html>\n");
            context.Response.OutputStream.BeginWrite(buffer, 0, buffer.Length, result =>
            {
                var response = (HttpListenerResponse)asyncResult.AsyncState;
                buffer = Encoding.UTF8.GetBytes("<html lang='en'>\n");
                response!.OutputStream.BeginWrite(buffer, 0, buffer.Length, result =>
                {
                    response = (HttpListenerResponse)asyncResult.AsyncState;
                    buffer = Encoding.UTF8.GetBytes("<head><meta charset='utf-8'><title>Example HTTP server</title></head>\n");
                    response!.OutputStream.BeginWrite(buffer, 0, buffer.Length, result =>
                    {
                        response = (HttpListenerResponse)asyncResult.AsyncState;
                        buffer = Encoding.UTF8.GetBytes("<body><p>Example HTTP server</p></body>\n");
                        response!.OutputStream.BeginWrite(buffer, 0, buffer.Length, result =>
                        {
                            response = (HttpListenerResponse)asyncResult.AsyncState;
                            buffer = Encoding.UTF8.GetBytes("</html>");
                            response!.OutputStream.BeginWrite(buffer, 0, buffer.Length, result =>
                            {
                                response = (HttpListenerResponse)asyncResult.AsyncState;
                                response!.OutputStream.Close();
                            }, response);
                        }, response);
                    }, response);
                }, response);
            }, context.Response);
        }
        else
        {
            context.Response.StatusCode = 404;
            context.Response.OutputStream.Close();
        }
    }
    else
    {
        context.Response.StatusCode = 405;
        context.Response.OutputStream.Close();
    }
}
```

async/await

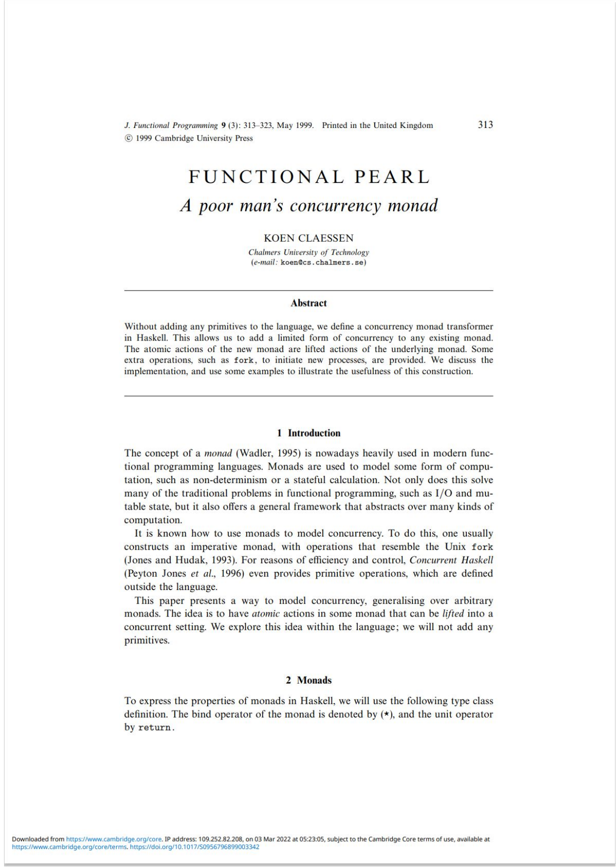
```
private static async void GetContextAsync(HttpListener listener)
{
    await Task.Yield();
    var context = await listener.GetContextAsync();
    GetContextAsync(listener);
    await Console.Out.WriteLineAsync($"{context.Request.HttpMethod} {context.Request.RawUrl}");

    if (context.Request.HttpMethod == "GET")
    {
        if (context.Request.RawUrl == "/")
        {
            await Task.Delay(100);
            context.Response.StatusCode = 200;

            await using var writer = new StreamWriter(context.Response.OutputStream);
            await writer.WriteLineAsync("<!DOCTYPE html>");
            await writer.WriteLineAsync("<html lang='en' xmlns='http://www.w3.org/1999/xhtml'>");
            await writer.WriteLineAsync("  <head>");
            await writer.WriteLineAsync("  <meta charset='utf-8' />");
            await writer.WriteLineAsync("  <title>Example HTTP server</title>");
            await writer.WriteLineAsync("  </head>");
            await writer.WriteLineAsync("  <body>");
            await writer.WriteLineAsync("    <p>Example HTTP server</p>");
            await writer.WriteLineAsync("  </body>");
            await writer.WriteLineAsync("</html>");
        }
        else
        {
            context.Response.StatusCode = 404;
            context.Response.OutputStream.Close();
        }
    }
    else
    {
        context.Response.StatusCode = 405;
        context.Response.OutputStream.Close();
    }
}
```

Было/стало

```
static void AsyncProcessRequest(IAsyncResult asyncResult)
{
    var listener = (HttpListener)asyncResult.AsyncState;
    listener!.BeginGetContext(AsyncProcessRequest, listener);
    var context = listener.EndGetContext(asyncResult);

    if (context.Request.HttpMethod == "GET")
    {
        if (context.Request.RawUrl == "/")
        {
            context.Response.StatusCode = 200;
            var buffer = Encoding.UTF8.GetBytes("<!DOCTYPE html>\n");
            context.Response.OutputStream.BeginWrite(buffer, 0, buffer.Length, result =>
            {
                var response = (HttpListenerResponse)asyncResult.AsyncState;
                buffer = Encoding.UTF8.GetBytes("<html lang='en'>\n");
                response!.OutputStream.BeginWrite(buffer, 0, buffer.Length, result =>
                {
                    response = (HttpListenerResponse)asyncResult.AsyncState;
                    buffer = Encoding.UTF8.GetBytes("<head><meta charset='utf-8'><title>Example HTTP server</title></head>\n");
                    response!.OutputStream.BeginWrite(buffer, 0, buffer.Length, result =>
                    {
                        response = (HttpListenerResponse)asyncResult.AsyncState;
                        buffer = Encoding.UTF8.GetBytes("<body><p>Example HTTP server</p></body>\n");
                        response!.OutputStream.BeginWrite(buffer, 0, buffer.Length, result =>
                        {
                            response = (HttpListenerResponse)asyncResult.AsyncState;
                            buffer = Encoding.UTF8.GetBytes("</html>");
                            response!.OutputStream.BeginWrite(buffer, 0, buffer.Length, result =>
                            {
                                response = (HttpListenerResponse)asyncResult.AsyncState;
                                response!.OutputStream.Close();
                            }, response);
                        }, response);
                    }, response);
                }, response);
            }, context.Response);
        }
        else
        {
            context.Response.StatusCode = 404;
            context.Response.OutputStream.Close();
        }
    }
    else
    {
        context.Response.StatusCode = 405;
        context.Response.OutputStream.Close();
    }
}
```

```
private static async void GetContextAsync(HttpListener listener)
{
    await Task.Yield();
    var context = await listener.GetContextAsync();
    GetContextAsync(listener);
    await Console.Out.WriteLineAsync($"{context.Request.HttpMethod} {context.Request.RawUrl}");

    if (context.Request.HttpMethod == "GET")
    {
        if (context.Request.RawUrl == "/")
        {
            await Task.Delay(100);
            context.Response.StatusCode = 200;

            await using var writer = new StreamWriter(context.Response.OutputStream);
            await writer.WriteLineAsync("<!DOCTYPE html>");
            await writer.WriteLineAsync("<html lang='en' xmlns='http://www.w3.org/1999/xhtml'>");
            await writer.WriteLineAsync("  <head>");
            await writer.WriteLineAsync("  <meta charset='utf-8' />");
            await writer.WriteLineAsync("  <title>Example HTTP server</title>");
            await writer.WriteLineAsync("  </head>");
            await writer.WriteLineAsync("  <body>");
            await writer.WriteLineAsync("    <p>Example HTTP server</p>");
            await writer.WriteLineAsync("  </body>");
            await writer.WriteLineAsync("</html>");
        }
        else
        {
            context.Response.StatusCode = 404;
            context.Response.OutputStream.Close();
        }
    }
    else
    {
        context.Response.StatusCode = 405;
        context.Response.OutputStream.Close();
    }
}
```

A poor man’s concurrency monad



J. Functional Programming **9** (3): 313–323, May 1999. Printed in the United Kingdom

FUNCTIONAL PEARL

A poor man’s concurrency monad

KOEN CLAESSEN
Chalmers University of Technology
(*e-mail:* koen@cs.chalmers.se)

Abstract

Without adding any primitives to the language, we define a concurrency monad transformer in Haskell. This allows us to add a limited form of concurrency to any existing monad. The atomic actions of the new monad are lifted actions of the underlying monad. Some extra operations, such as fork, to initiate new processes, are provided. We discuss the implementation, and use some examples to illustrate the usefulness of this construction.

1 Introduction

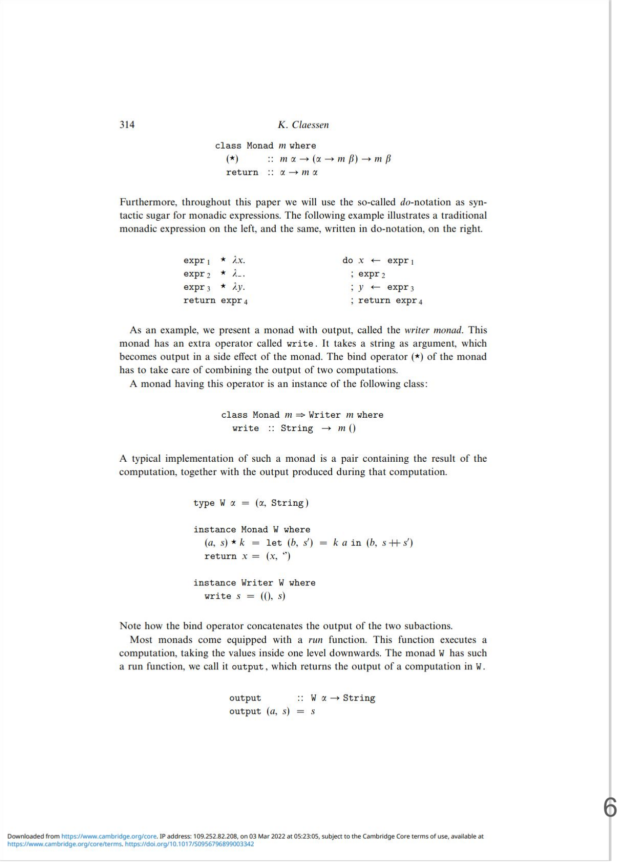
The concept of a *monad* (Wadler, 1995) is nowadays heavily used in modern functional programming languages. Monads are used to model some form of computation, such as non-determinism or a stateful calculation. Not only does this solve many of the traditional problems in functional programming, such as I/O and mutable state, but it also offers a general framework that abstracts over many kinds of computation.

It is known how to use monads to model concurrency. To do this, one usually constructs an imperative monad, with operations that resemble the Unix fork (Jones and Hudak, 1993). For reasons of efficiency and control, *Concurrent Haskell* (Peyton Jones *et al.*, 1996) even provides primitive operations, which are defined outside the language.

This paper presents a way to model concurrency, generalising over arbitrary monads. The idea is to have *atomic* actions in some monad that can be *lifted* into a concurrent setting. We explore this idea within the language; we will not add any primitives.

2 Monads

To express the properties of monads in Haskell, we will use the following type class definition. The bind operator of the monad is denoted by (⋆), and the unit operator by return.



```
class Monad m where
  (⋆)      :: m α → (α → m β) → m β
  return   :: α → m α
```

Furthermore, throughout this paper we will use the so-called *do*-notation as syntactic sugar for monadic expressions. The following example illustrates a traditional monadic expression on the left, and the same, written in do-notation, on the right.

```
expr₁  ⋆ λx.                        do x ← expr₁
expr₂  ⋆ λ_.                         ; expr₂
expr₃  ⋆ λy.                         ; y ← expr₃
return expr₄                         ; return expr₄
```

As an example, we present a monad with output, called the *writer monad*. This monad has an extra operator called write. It takes a string as argument, which becomes output in a side effect of the monad. The bind operator (⋆) of the monad has to take care of combining the output of two computations.

A monad having this operator is an instance of the following class:

```
class Monad m ⇒ Writer m where
  write  :: String  →  m ()
```

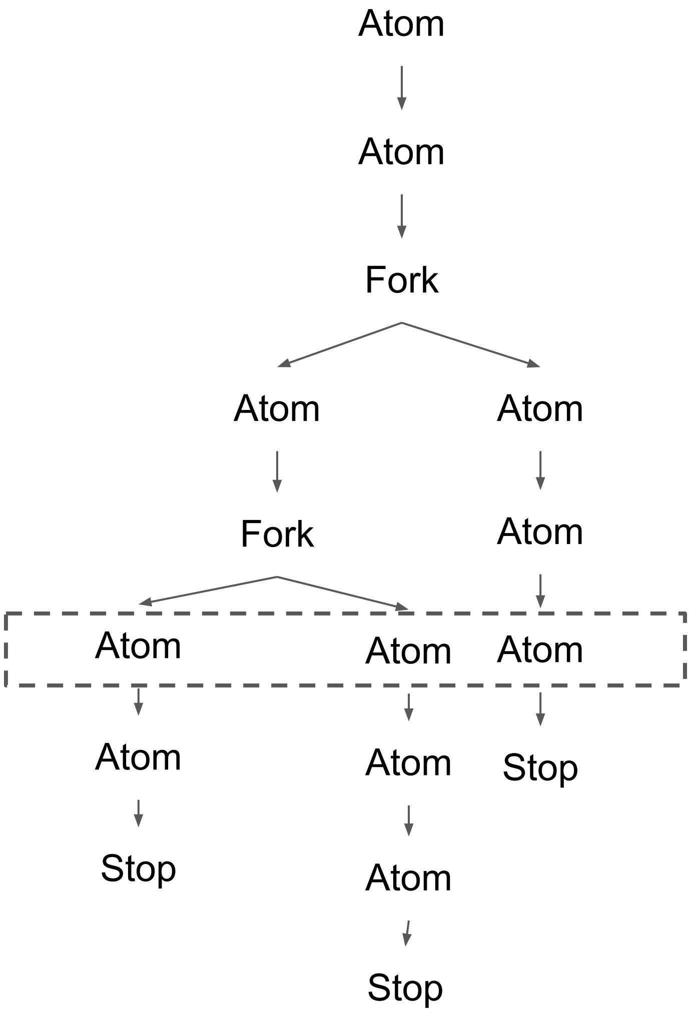
A typical implementation of such a monad is a pair containing the result of the computation, together with the output produced during that computation.

```
type W α = (α, String)

instance Monad W where
  (a, s) ⋆ k  =  let (b, s′) = k a in (b, s ++ s′)
  return x = (x, “)

instance Writer W where
  write s = ((), s)
```

Note how the bind operator concatenates the output of the two subactions.

Most monads come equipped with a *run* function. This function executes a computation, taking the values inside one level downwards. The monad W has such a run function, we call it output, which returns the output of a computation in W.

```
output          :: W α → String
output (a, s)   = s
```



Atom

Atom

Fork

Atom Atom

Fork Atom

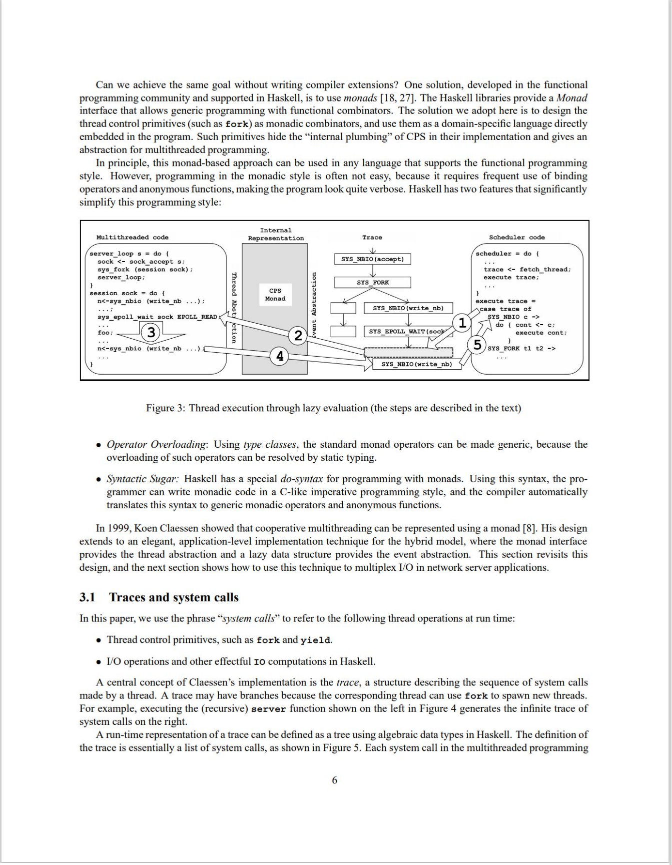
Atom Atom Atom

Atom Atom Stop

Stop Atom

Stop

Combining Events And Threads For Scalable Network



Can we achieve the same goal without writing compiler extensions? One solution, developed in the functional programming community and supported in Haskell, is to use *monads* [18, 27]. The Haskell libraries provide a *Monad* interface that allows generic programming with functional combinators. The solution we adopt here is to design the thread control primitives (such as **`fork`**) as monadic combinators, and use them as a domain-specific language directly embedded in the program. Such primitives hide the "internal plumbing" of CPS in their implementation and gives an abstraction for multithreaded programming.

In principle, this monad-based approach can be used in any language that supports the functional programming style. However, programming in the monadic style is often not easy, because it requires frequent use of binding operators and anonymous functions, making the program look quite verbose. Haskell has two features that significantly simplify this programming style:

```
server_loop s = do {
  sock <- sock_accept s;
  sys_fork (session sock);
  server_loop;
}
session sock = do {
  n<-sys_nbio (write_nb ...);
  ...;
  sys_epoll_wait sock EPOLL_READ;
  ...
  foo;
  ...
  n<-sys_nbio (write_nb ...);
  ...
}
```

```
scheduler = do {
  ...
  trace <- fetch_thread;
  execute trace;
  ...
}
execute trace =
 case trace of
  SYS_NBIO c ->
   do { cont <- c;
        execute cont;
      }
  SYS_FORK t1 t2 ->
   ...
```

Figure 3: Thread execution through lazy evaluation (the steps are described in the text)

- *Operator Overloading*: Using *type classes*, the standard monad operators can be made generic, because the overloading of such operators can be resolved by static typing.
- *Syntactic Sugar:* Haskell has a special *do-syntax* for programming with monads. Using this syntax, the programmer can write monadic code in a C-like imperative programming style, and the compiler automatically translates this syntax to generic monadic operators and anonymous functions.

In 1999, Koen Claessen showed that cooperative multithreading can be represented using a monad [8]. His design extends to an elegant, application-level implementation technique for the hybrid model, where the monad interface provides the thread abstraction and a lazy data structure provides the event abstraction. This section revisits this design, and the next section shows how to use this technique to multiplex I/O in network server applications.

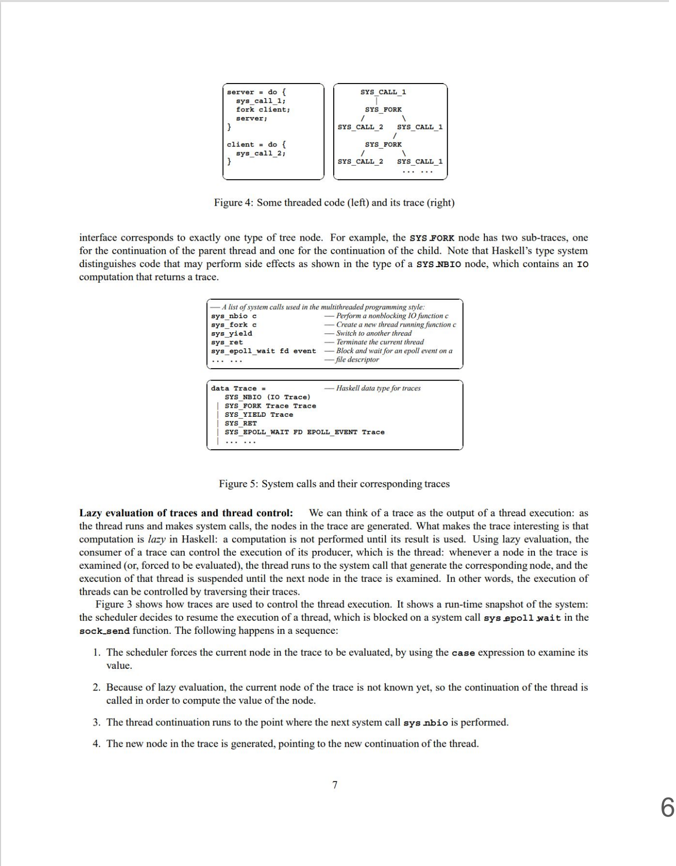
3.1 Traces and system calls

In this paper, we use the phrase "*system calls*" to refer to the following thread operations at run time:

- Thread control primitives, such as **`fork`** and **`yield`**.
- I/O operations and other effectful **`IO`** computations in Haskell.

A central concept of Claessen's implementation is the *trace*, a structure describing the sequence of system calls made by a thread. A trace may have branches because the corresponding thread can use **`fork`** to spawn new threads. For example, executing the (recursive) **`server`** function shown on the left in Figure 4 generates the infinite trace of system calls on the right.

A run-time representation of a trace can be defined as a tree using algebraic data types in Haskell. The definition of the trace is essentially a list of system calls, as shown in Figure 5. Each system call in the multithreaded programming



```
server = do {
  sys_call_1;
  fork client;
  server;
}

client = do {
  sys_call_2;
}
```

```
      SYS_CALL_1
         |
       SYS_FORK
      /        \
SYS_CALL_2   SYS_CALL_1
               /
       SYS_FORK
      /        \
SYS_CALL_2   SYS_CALL_1
               ... ...
```

Figure 4: Some threaded code (left) and its trace (right)

interface corresponds to exactly one type of tree node. For example, the **`SYS_FORK`** node has two sub-traces, one for the continuation of the parent thread and one for the continuation of the child. Note that Haskell's type system distinguishes code that may perform side effects as shown in the type of a **`SYS_NBIO`** node, which contains an **`IO`** computation that returns a trace.

```
— A list of system calls used in the multithreaded programming style:
sys_nbio c                   — Perform a nonblocking IO function c
sys_fork c                   — Create a new thread running function c
sys_yield                    — Switch to another thread
sys_ret                      — Terminate the current thread
sys_epoll_wait fd event      — Block and wait for an epoll event on a
... ...                      — file descriptor
```

```
data Trace =                 — Haskell data type for traces
    SYS_NBIO (IO Trace)
  | SYS_FORK Trace Trace
  | SYS_YIELD Trace
  | SYS_RET
  | SYS_EPOLL_WAIT FD EPOLL_EVENT Trace
  | ... ...
```

Figure 5: System calls and their corresponding traces

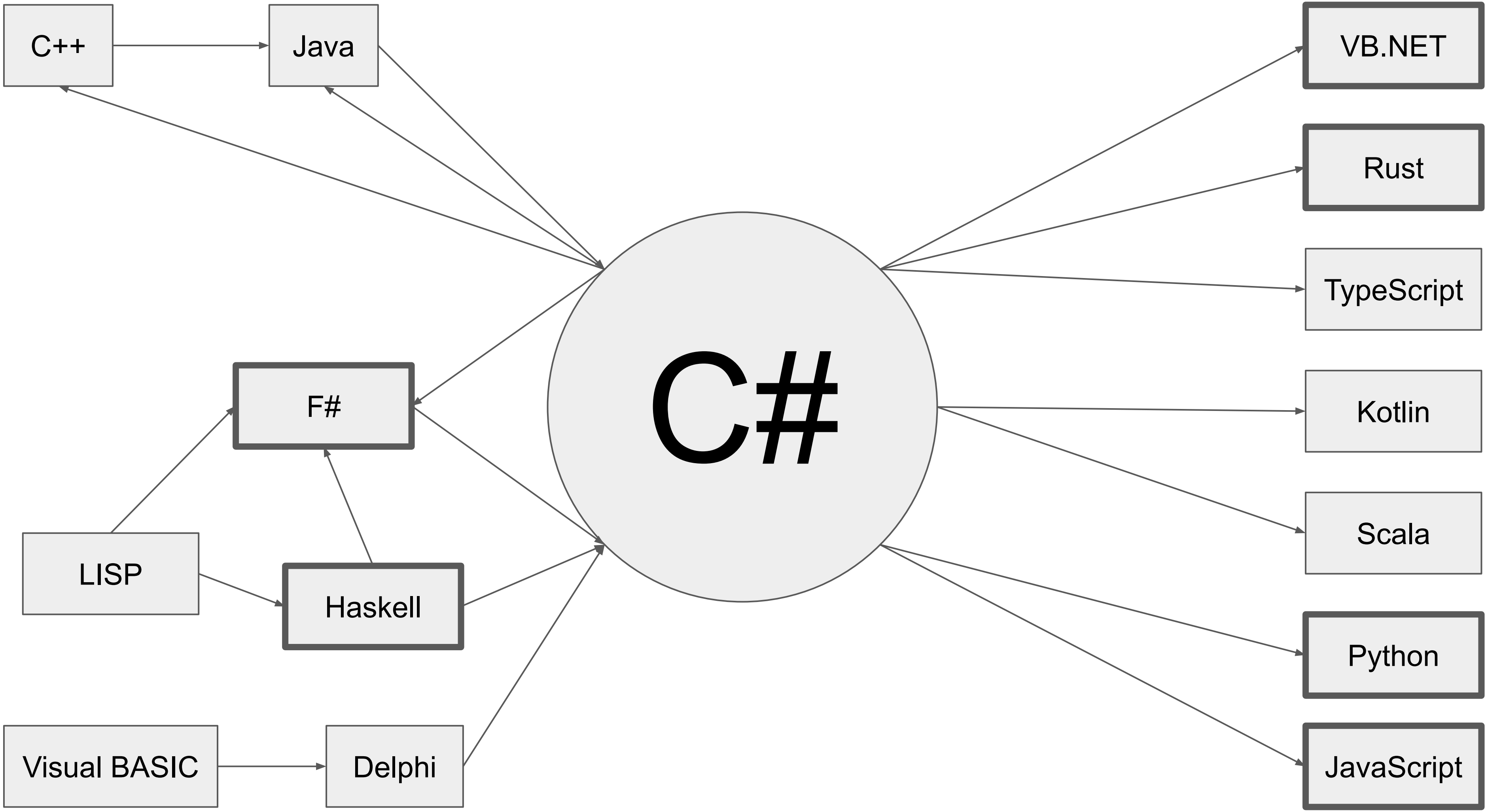
Lazy evaluation of traces and thread control: We can think of a trace as the output of a thread execution: as the thread runs and makes system calls, the nodes in the trace are generated. What makes the trace interesting is that computation is *lazy* in Haskell: a computation is not performed until its result is used. Using lazy evaluation, the consumer of a trace can control the execution of its producer, which is the thread: whenever a node in the trace is examined (or, forced to be evaluated), the thread runs to the system call that generate the corresponding node, and the execution of that thread is suspended until the next node in the trace is examined. In other words, the execution of threads can be controlled by traversing their traces.

Figure 3 shows how traces are used to control the thread execution. It shows a run-time snapshot of the system: the scheduler decides to resume the execution of a thread, which is blocked on a system call **`sys_epoll_wait`** in the **`sock_send`** function. The following happens in a sequence:

1. The scheduler forces the current node in the trace to be evaluated, by using the **`case`** expression to examine its value.
2. Because of lazy evaluation, the current node of the trace is not known yet, so the continuation of the thread is called in order to compute the value of the node.
3. The thread continuation runs to the point where the next system call **`sys_nbio`** is performed.
4. The new node in the trace is generated, pointing to the new continuation of the thread.

Introducing F# Asynchronous Workflows

```
let AsyncHttp(url:string) = async {
    let req = WebRequest.Create(url)

    let! rsp = req.GetResponseAsync()

    use stream = rsp.GetResponseStream()
    use reader = new System.IO.StreamReader(stream)

    return reader.ReadToEnd()
}
```


C++
Java
VB.NET
Rust
TypeScript
F#
C#
Kotlin
Scala
LISP
Haskell
Python
Visual BASIC
Delphi
JavaScript

Литература

- *Харольд Абельсон и Джеральд Сассман*. Структура и интерпретация компьютерных программ
- *Алан Купер*. Психбольница в руках пациентов
- *Christopher Strachey*. Fundamental Concepts in Programming Languages
- *Koen Claessen*. A Poor Man’s Concurrency Monad
- *Peng Li, Stephan A. Zdancewic*. Combining Events And Threads For Scalable Network Services
- *Tomas Petricek*. Asynchronous C# and F#
- SySharp: https://github.com/markshevchenko/sysharp

Идет набор в Школу бэкенд-разработки!

Подать заявку и выполнить вступительное задание нужно **до 23 июня 23.59 по московскому времени**

- Обучение в школе проходит на двух треках: Python и Java
- Студентов школы ждут два этапа: онлайн с лекциями и практическими занятиями и очный в Москве с реализацией проектов в командах (Яндекс оплатит билеты и проживание студентам из других городов)
- Обучение **полностью бесплатное**

Заключение

Марк Шевченко
mark-progmsk@yandex-team.ru
https://markshevchenko.pro
@markshevchenko

- Свойства.
- События и делегаты.
- Полиморфизм.
- Вывод типов.
- Лямбды.
- Деревья выражений.
- Ленивые вычисления.
- Оператор GOTO.
- Асинхронный код.