# Розробка інтерпретованої мови програмування для опису даних та імперативної логіки у безпечних програмних середовищах

◎ Студент БІКС-22 - Шпак М. Р.

# Структура виступу по обраній темі

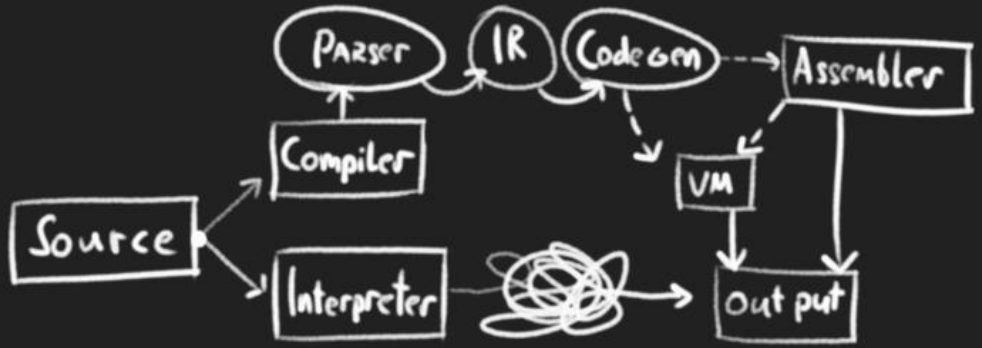
+ Причина вибору теми дипломного проектування
+ Огляд сфери що досліджується
+ Вилявлені проблеми
+ Методи вирішення
+ Визначення вимог
+ Розроблений прототип - проекти Duct та HCH
+ Технічні аспекти
+ Демонстація
+ Переваги та недоліки
+ Перспективи розвитку

# Чому мова програмування?

- Кібер Спеціаліст > Розробник ПЗ, в сфері знань програмування та інформаційних технологій.
- Неоднозначні думки в понятті “безпеки” в мовах програмування
- Популяризація Rust, Zig та мов що збираються для WASM.
- Мови програмування цікаві!

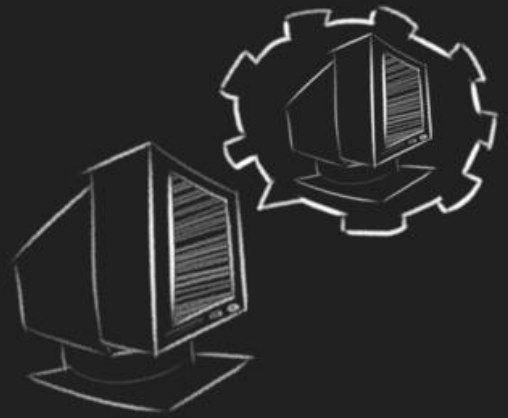
# Необхідна (до згадування) термінологія

- Високо- та низько- рівневість мови описує її наближеність до дійсної роботи комп'ютера.
- Системні мови програмування оперують над пам'ять, стеком та регістром, скриптові над змінними, функціями та мета-інформацією
- Компілятор ≈ Інтерпетатор



# Віртуалізація, інтерпретація, JIT.

- Віртуальна машина - симуляція комп'ютера в комп'ютері.
- Інтерпретатор - зчитування та обробка тексту в реальному часі.
- JIT(Just-In-Time compilation) - напів-системна мова програмування, суміш Віртуальної машини та збираємих мов.
- Компілятор ≈ Інтерпетатор



# Проблеми вказівників

- Вказівники є джерелом 70+% вразливостей пам’яті.
- Легко використовувати, важко зрозуміти.
- Вказівники складно відслідковувати, особливо в багато-потоковому середовищі.
- В системних мовах програмування після збірки втрачається інформація про природу даних в програмі.

# Метапрограмування

We weren't writing this code for our own amusement. We were a tiny startup, programming as hard as we could in order to put technical barriers between us and our competitors.

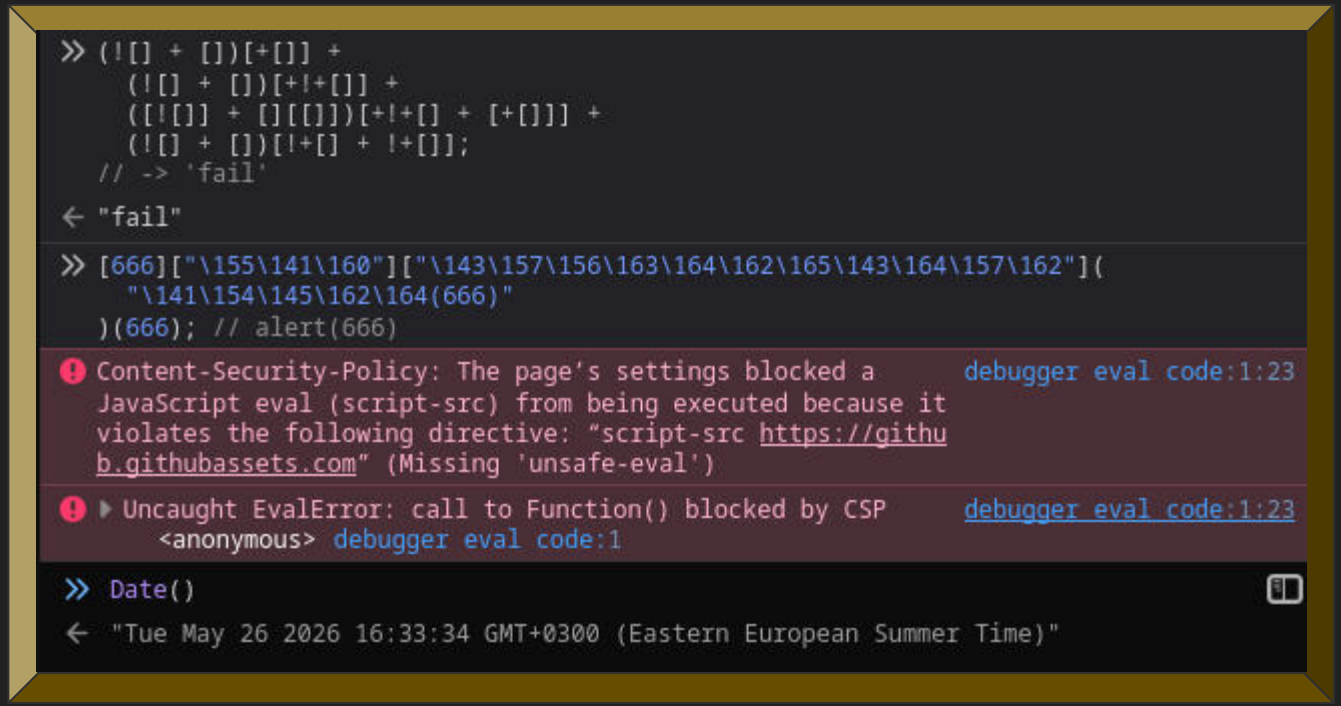
**Paul Graham** - What made Lisp different (2001)

# Занадто вільний синтаксис

- Звертання до змінних яких не існує
- Перевизначення функцій
- Відсутність констант та нумерацій (Enums)
- Інтерполяція строк та вільна (безконтрольна) конвертація даних
- Перевантаження операторів (Operator overloading)

```
  [~] › lua
Lua 5.5.0  Copyright (C) 1994-2025 Lua.org, PUC-Rio
> print("I want to be free")
I want to be free
> p = print print = function(...) p("you are hacked!") end
> print("I want to break free")
you are hacked!
```

# ????

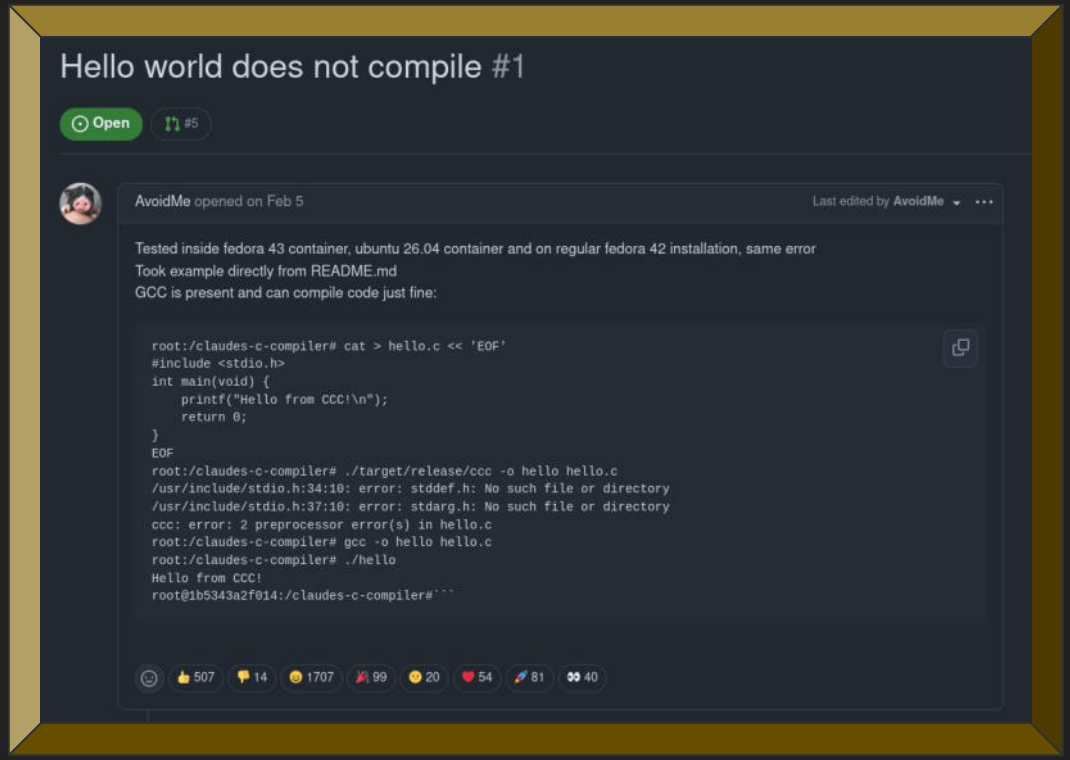

» (![] + [])[+[]] +
([] + [])[+!+[]] +
([![]] + [][[]])[+!+[] + [+[]]] +
(![] + [])[!+[] + !+[]];
// -> 'fail'
← "fail"
» [666]["\155\141\160"]["\143\157\156\163\164\162\165\143\164\157\162"](
"\141\154\145\162\164(666)"
)(666); // alert(666)
Content-Security-Policy: The page's settings blocked a JavaScript eval (script-src) from being executed because it violates the following directive: "script-src https://github.githubassets.com" (Missing 'unsafe-eval')
debugger eval code:1:23
Uncaught EvalError: call to Function() blocked by CSP
<anonymous> debugger eval code:1
debugger eval code:1:23
» Date()
← "Tue May 26 2026 16:33:34 GMT+0300 (Eastern European Summer Time)"

# Тенденції в проектуванні ПЗ

- Складність системи прямо-пропорційна її довговічності.
- Сучасна рекламна метрика це кількість строк коду а не його якість.
- Чим менше складність системи, тим менша вірогідність не знайти в ній недоліки.
- Безпека ядра Linux не в елегантності дизайну а в кількості очей над ним.

| | | |
|---|---|---|
| malloc vs ZGC | 6k loc | 5300+k loc |
| lua vs python | 14k loc | 660k loc |
| tcc vs gcc | 76k loc | 15000+k loc |

# Можливо застосувати штучний інтелект?




https://github.com/anthropics/claudes-c-compiler/issues/1

# Замість вказівників - абстракція

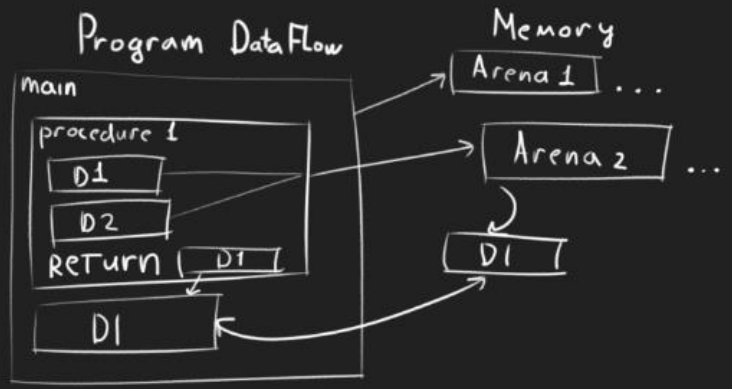
+ Вказівники замінено на типи даних які будуються на них.
+ Використання метаданих для інформації (Runtime Reflections).
+ Робота з комплексними даними має бути значно обмежена.

```
var
    pointer         :   *i32    ,
    array           :   [10]i32 ,
    array_pointer   :   [*]i32  ,
    array_slice     :   []i32   ;
```

Приклад типізації вказівників в Zig

# Спрощення керування пам'яттю

- Керування пам'яттю не є складною задачею.
- Сміттєзбирач є елегантним рішенням не існуючої проблеми.
- Стає обмежувальним фактором при проблемній архітектурі рішення.
- Потребує розуміння природи походження вирішуваної проблеми або дотримання строгих правил.



# Строгий синтаксис, детальний аналіз

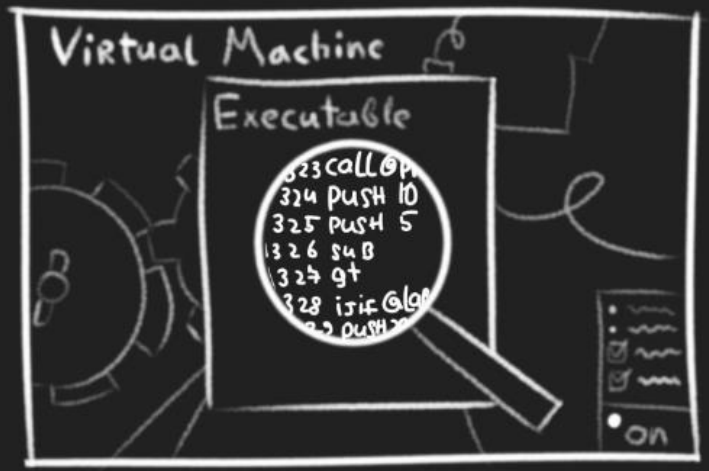
+ Простий синтаксис легко піддається аналізу
+ Аналіз - гарантія відсутності певних проблем, перевірка людського фактору
+ Синтаксис є зображенням правил мови, що змушує до співпраці а не боротьби з нею.

# Відсутність метапрограмування

+ Менший об'єм генеруємого програмного коду для аналізу
+ Проблеми легше знайти
+ Відсутність вектору атак через відсутність будь-якої типізації
+ Проблема для вирішення філософами а не інженерами чи кібераналітиками.
+ Краща швидкість роботи

# Віртуалізація

+ Забезпечує додатковий рівень інкапсуляції даних між користувацьким скриптом та окреструючим ПЗ
+ Надає змогу моніторингу роботи ПЗ в реальному часі
+ Розбиває процес запуску програми на простіші до виконання кроки



# Компілятор Duct

- + Однокроковий прохід.
- + Коротко-контекстний аналіз.
- + Чергова токенізації, технологія блокової буферизації.
- + Рекурсивно-поглиблювальний парсер.
- + Збирає до мови ассемблера ВМ Duct.
- + Аналіз символів перед їх використанням.

```
> function:fib
 > block:
  > statement:
   > if:
    > expr
          > value: number
      > compare |
          > value: 1
    > block:
     > statement:
      > expr
            > value: number
  > statement:
   > expr
          > fncall 'fib'
           > expr
                 > value: number
              > term |
                   > value: 1
         > value: fib
      > term |
          > fncall 'fib'
           > expr
                 > value: number
              > term |
                   > value: 2
         > value: fib
```

```
fn fib number
        load number
        push 1
        lte
        ijif 4
        pop
        load number
        ret
        jmp label000000
        pop
        @label000000
        load number
        push 1
        sub
        call fib
        load number
        push 2
        sub
        call fib
        add
        ret
endfn
```

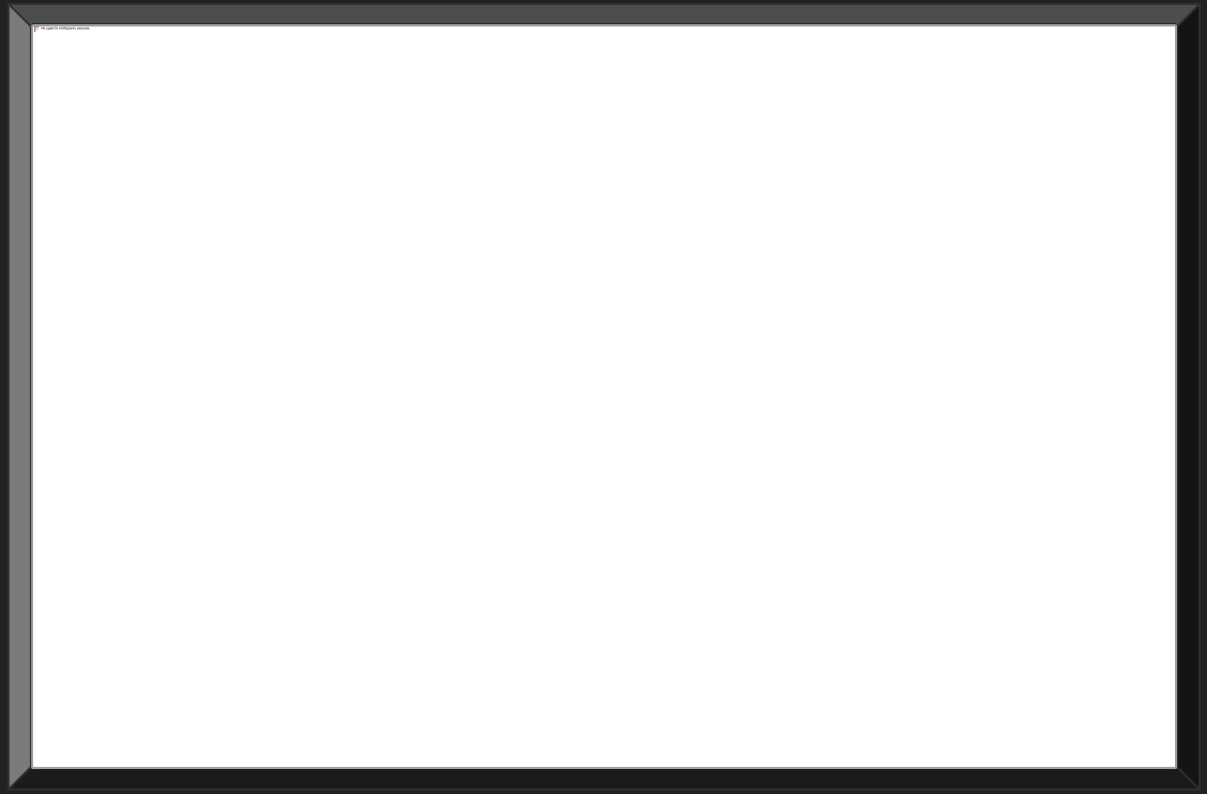
# Віртуальна Машина Duct

- + Стекова архітектура
- + Високо-рівневий підхід - моно поліморфність даних.
- + Пакування функцій для близькості в пам'яті.
- + Мінімальний набір інструкцій.
- + Ассемблер та "Розбирач" включено
- + Система для покрокового логування інструкцій.
- + API для контролю доступу до зовнішнього середовища.

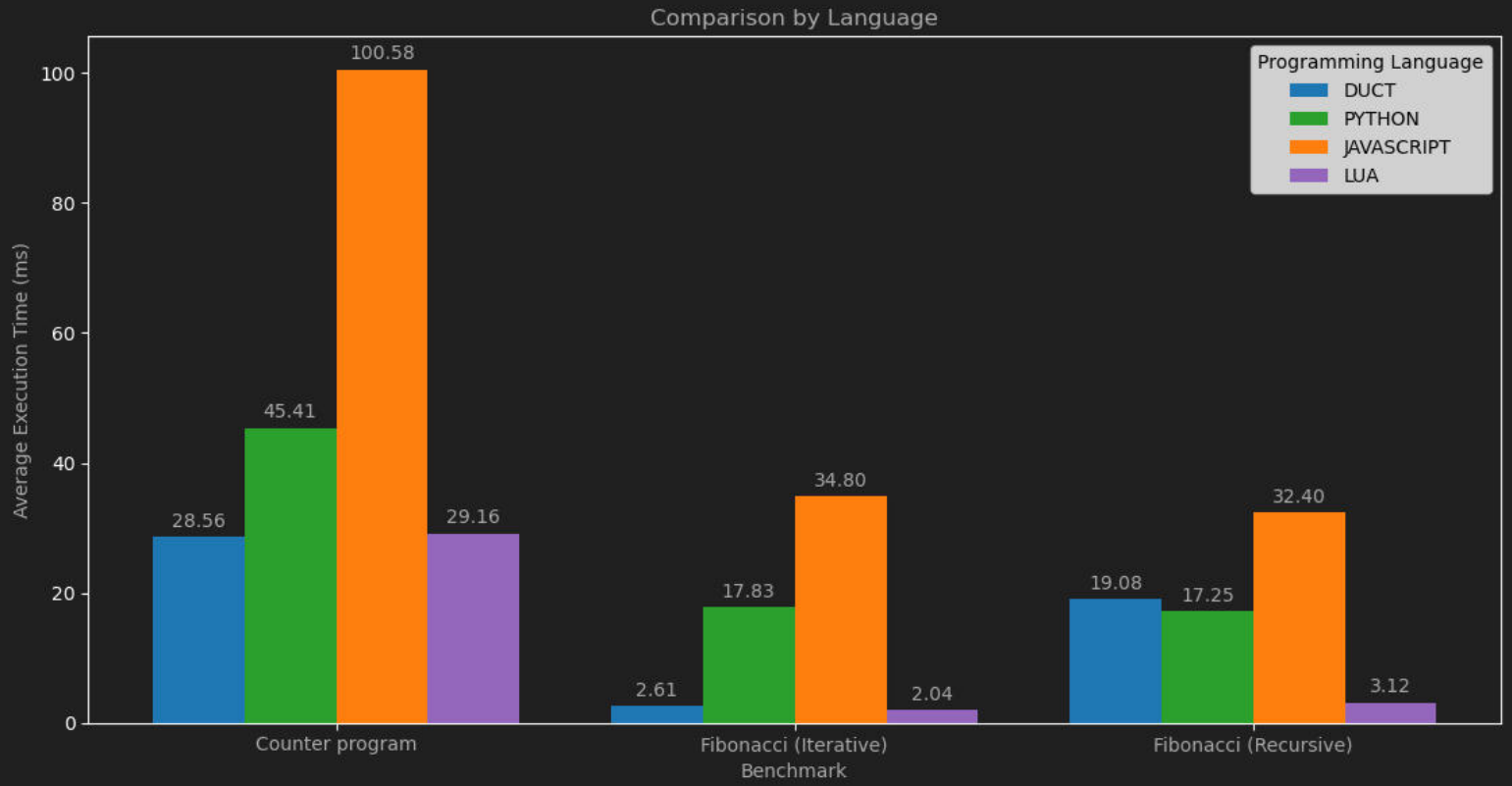
```
# Emmiting bytecode for 'main':
000000  push s(this is: )
000001  wrt
000002  pop
000003  push B.true
000004  wrt
000005  pop
        ARGS: [ ]
F:0000   000000 > push s(this is: )
--------------------------
F:0000   000001 > wrt
000000  $,      "this is: ";
--------------------------
F:0000   000002 > pop
000000  $,      "this is: ";
--------------------------
F:0000   000003 > push B.true
--------------------------
F:0000   000004 > wrt
000000  $,      true;
--------------------------
F:0000   000005 > pop
000000  $,      true;
--------------------------
        Data stack pointer : 0
this is: true⏎
```

# Демонстрація скриптів Duct

- + Демонстрація рекурсії
- + Демонстрація циклів та бранчування
- + Робочий оператор print
- + CLI для взаємодії
- + Інструменти відладки



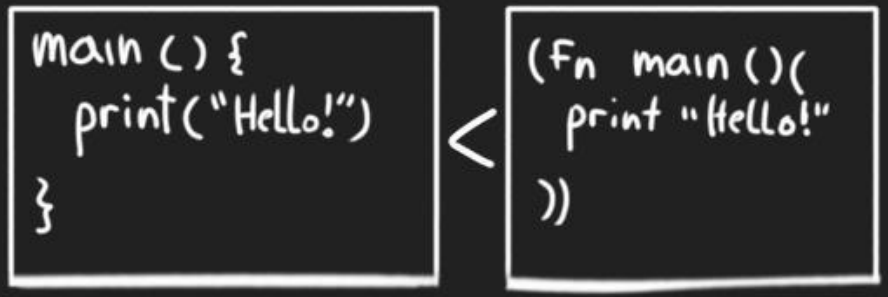
# Швидкість, Простота




Висновки - Переваги

# Імперативний підхід < Декларативний підхід

- Безпечніший метод організації використовує функції без побічних ефектів
- Дані та логіка для їх отримання більше притаманні декларативному мисленню
- Потрібен менший розмір кодової бази для забезпечення однакового функціоналу



# Перспективи розвитку

- Рефакторинг кодової бази
- Завершити повний функціонал мови - Сети, Строки, Масиви, Списки
- Стандартна бібліотека
- Зміна парадигми
  - АБО Декілька форм написання (Lisp-like, C-like)
- Свій маскот :)