# *Анализ и оптимизация на софтуерни приложения*

Александър Пенев

Васил Василев

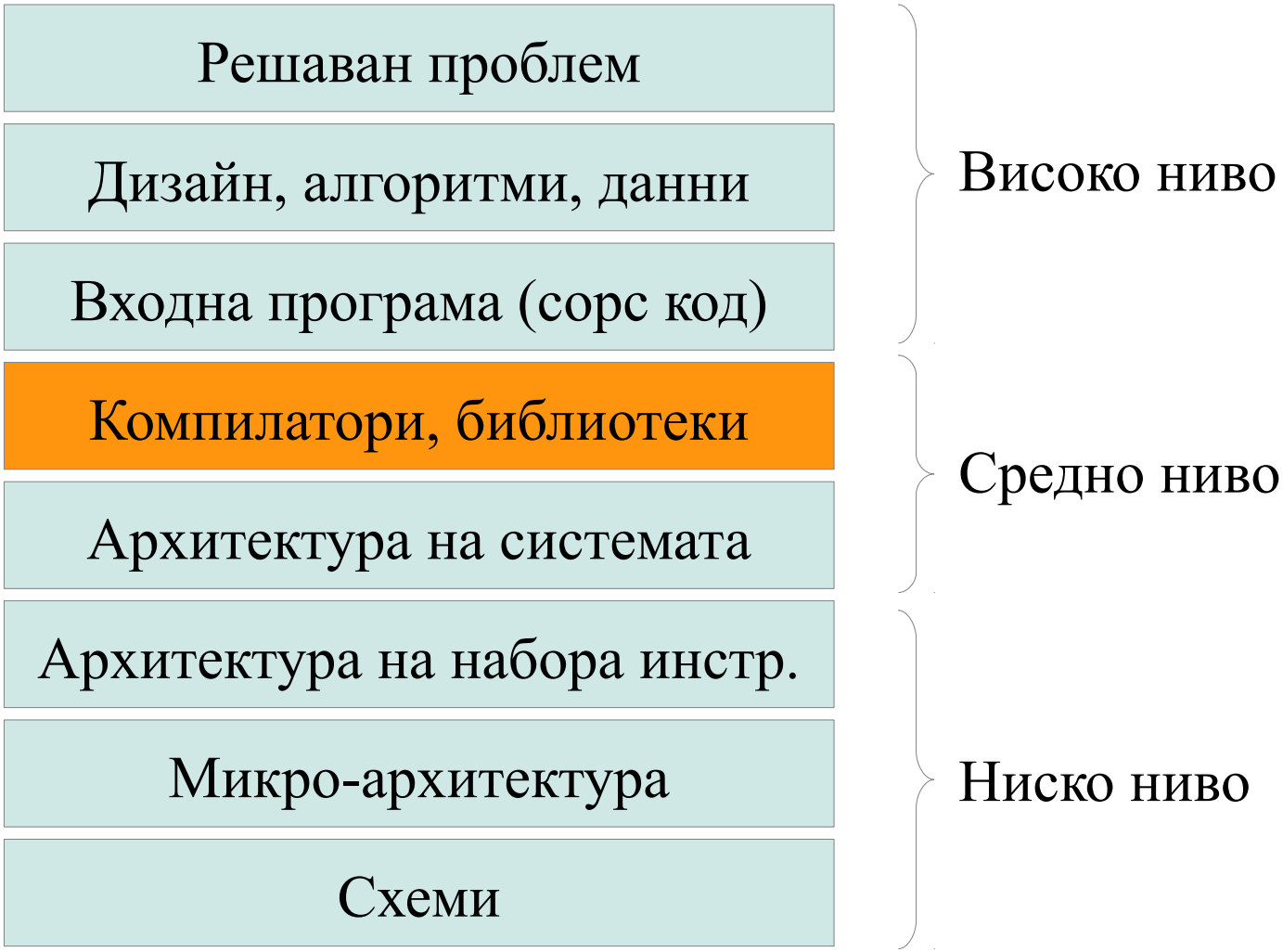
## Въведение в компилаторите

(от гледна точка на производителността)

# *Съдържание*

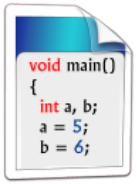
1. Видове транслатори
2. Фази по време на компилация
3. Оптимизационна фаза

# *Къде сме в момента?*



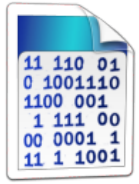
# *Компилатор или интерпретатор*

**Компилатор** транслира до машинен код




*сорс код*

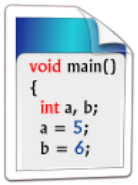
→ лексик. анализ → синтакт. анализ → ... → генерация на код → линкер →




*машинен код*

→ loader

**Интерпретатор** изпълнява сорс кода “директно”



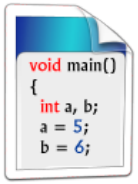

*сорс код*

→ лексик. анализ → синтакт. анализ →

*интерпретация*

- Операторите в цикъл се анализират отново и отново

**Хибриден компилатор** интерпретира междинен код




*сорс код*

→ ...компилатор... →


*междинен код (напр. Common Intermediate Language (CIL))*

→ ВИМ

- Сорс кода се транслира до код за виртуално-изчислителна машина (ВИМ)
- ВИМ интерпретира кода, симулирайки реална машина

# *Кратка история*

В миналото “загадка”, сега една от най-добре познатите области в информатиката

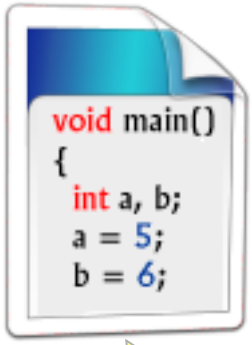
| | | |
|---|---|---|
| **1957** | Fortran | първи компилатори (аритм. изрази, изречения, процедури) |
| **1960** | Algol | първата явна дефиниция на език (граматики във форма на Бакус-Наур, блокова структура, рекурсии) |
| **1970** | Pascal | потребителски дефинирани типове, виртуални машини |
| **1985** | C++ | обектно-ориентиран, изключения, шаблони (templates) |
| **1995** | Java | just-in-time компилация |

***Разглеждат се само** *императивни езици*

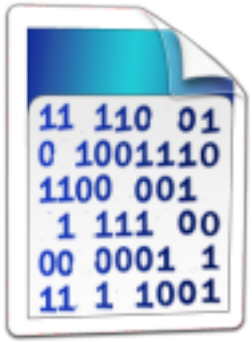
# *Какво представлява компилаторът*

*Език от високо ниво*

*Език от ниско ниво*








*Един (няколко) от езиците:*

- *C/C++/Objective-C*
- *C#/Java*
- *Pascal/Delphi*
- *...*

*Може да бъде:*

- *Машинен код (x86 и др.)*
- *LLVM bitcode*
- *CIL*
- *Java Bytecode*
- *...*

# *Защо компилаторите са важни?*

Защо трябва да ни интересуват компилаторите?
„Компилаторът е само инструмент...“

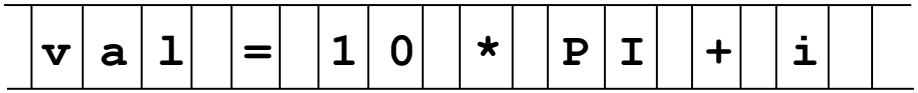
- Компилаторът не е само инструмент
- Той има отговорността да каже на компютъра какво Вие искате да му кажете/направите

Познаването на процеса на компилация помага на програмиста да съставя по-ефективни програми.

- Важно е да се знае какво може да направи компилатора за вас и **какво не**

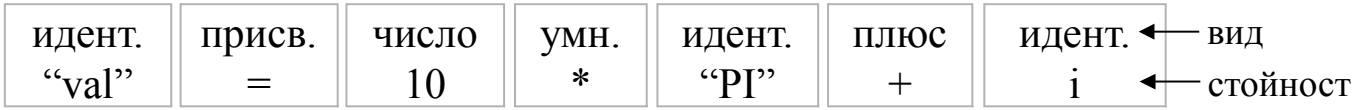
# *Динамична структура на компилатор*

Поток от символи



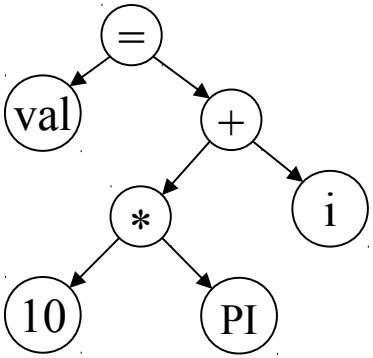

Лексикален анализ (скарниране)

Поток от лексеми



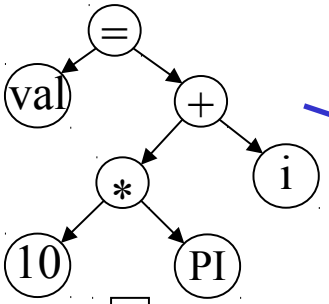

Синтактичен анализ (разпознаване)

Синтактично дърво



# *Динамична структура на компилатор*

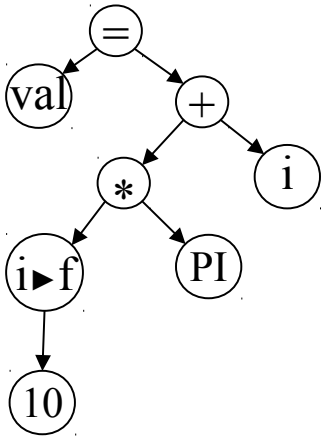
Синтактично дърво




Семантичен анализ

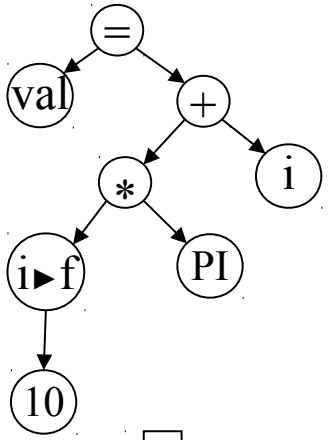
| | | | |
|---|---|---|---|
| 1 | val | float | ... |
| 2 | PI | float | ... |
| 3 | i | float | ... |
| ... | | | |

Синтактично дърво,
символна таблица, ...



# *Динамична структура на компилатор (пр.)*

Синтактично дърво




Маш. независима оптимизация

Генерация на междинен код

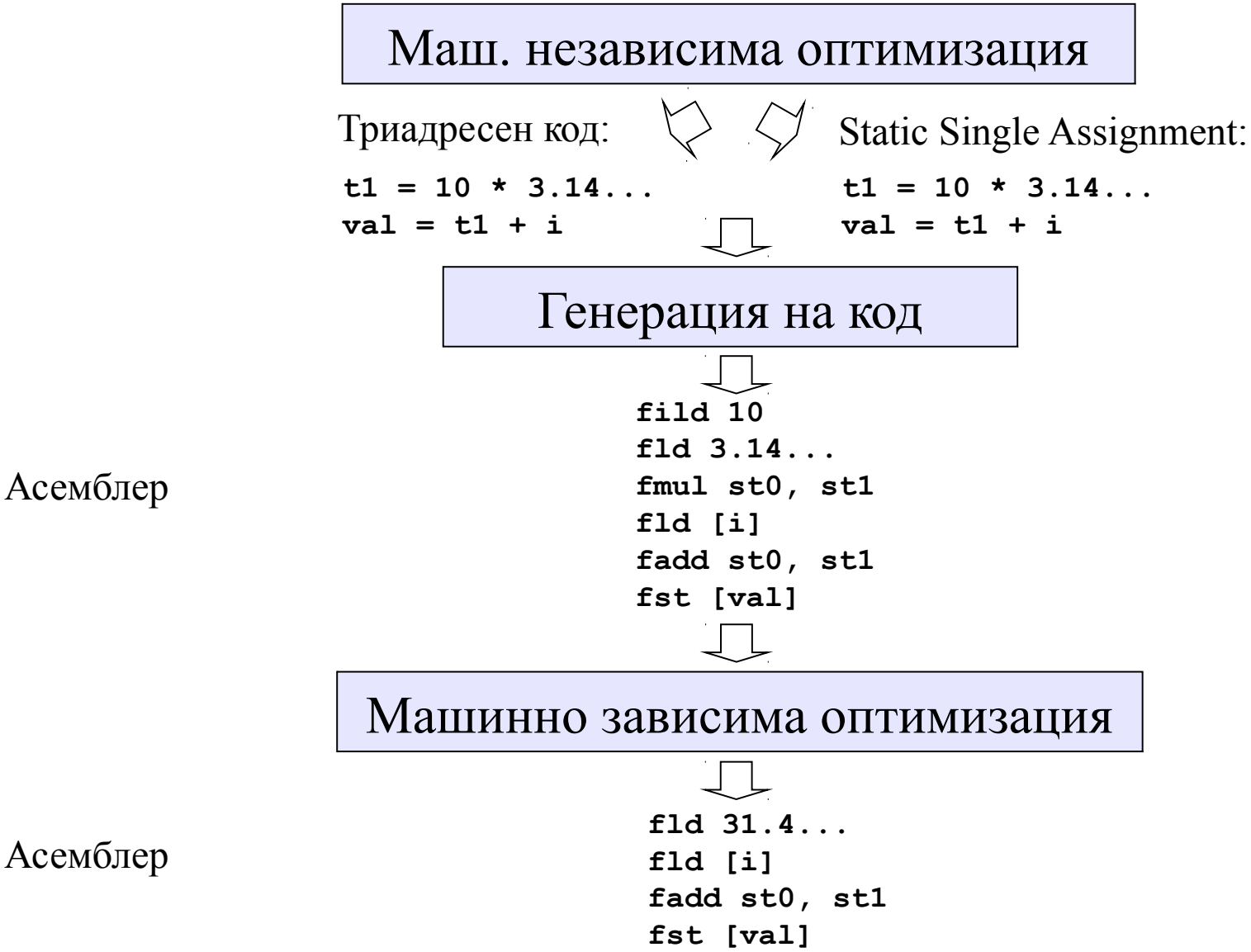
Триадресен код:

```
t0 = inttofloat(10);
t1 = t0 * PI
t0 = t1 + i
val = t0
```

Static Single Assignment:

```
t0 = inttofloat(10);
t1 = t0 * PI
t2 = t1 + i
val = t2
```

# *Динамична структура на компилатор (пр.)*



# *Оптимизационни фази*

Оптимизационните фази са:

- Машинно независими

  *Трансформации върху някое от междинните представяния на компилатора*

- Машинно зависими

  *Трансформации върху генерирания код, приложими за конкретни архитектури*

# *Стъпки в оптимизационната фаза*

- Анализ
  *Намират се проблемни места в програмата или междинното представяне*
- Идентификация
  *След като анализът е показал, че има проблемно място, се идентифицира мястото в програмата или представянето и се определя вида трансформация, която да се предприеме*
- Трансформация
  *Трансформацията преобразува кода в най-често в по-ефективен, еквивалентен на предишния, код*

# *Типове оптимизации*

- Peephole optimizations

  *Обикновено се извършват късно (в процеса на компилация), след като се генерира машинен код. Този тип оптимизации изследва няколко съседни инструкции и преценява дали може да ги замени с една инструкция или с по-малка последователност от инструкции.*

- Локални оптимизации

  *Те използват информацията само достъпна в тялото на целевата функция. Това намалява времената за анализ, но ограничава оптимизацията.*

- Междупроцедурни оптмизации

  *Тези оптимизации анализират целия код на програмата. Повечето събрана информация означава, че оптимизациите могат да бъдат по-ефективни в сравнение с локалните*

- Оптимизация на цикли

  *Тези оптимизации действат върху оператори за цикъл в езиците за програмиране (като for и while). Този тип може да постигне значително подобрение, защото най-често циклите изразходват много време при изпълнение*

# *Типични оптимизации*

- Развиване на константи
  *Опростяват се константните изрази като се извършват изчисленията по време на компилация*
- Разпространение на константи
  *Заместват се стойностите на известните константи в изразите*
- Елиминиране на мъртъв код
  *Премахва се кодът в програмата, който не може да се изпълни при никакви обстоятелства. Мъртвия код е два вика: недостижим код и мъртви присвоявания (записи)*
- Елиминиране на общи подизрази
  *Премахва общите подизрази като ги изважда като променлива и след това ги замества* с тази променлива
- Развиване на цикли
  *Редуцират се броя на итерациите в цикъла като се използват различни техники. Например копиране на тялото на цикъла няколко пъти и увеличаване на стъпката на цикъла*
- Заделяне на регистри
  *Използва максимално добре наличните регистри*

# *Фактори, указващи влияние на оптимизациите*

- Целевата машина

  *Много от решенията за прилагане на оптимизации се взимат като се има предвид целевата машина. Понякога е възможно тези фактори на целевата машина да се параметризират. Така компилаторът може да се използва да прилага оптимизации, специфични за различни целеви машини.*

- Архитектурата на целевата машина

  *Размер и тип на кеша (32 kB – 16MB).*

- Микроархитектурата на целевия процесор

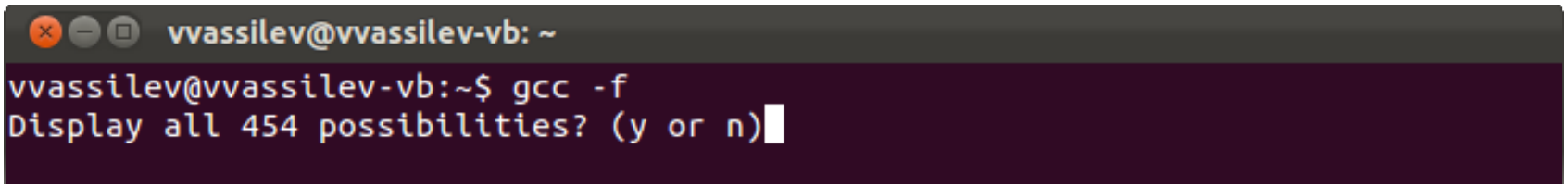
  *Зависи до определена степен от броя регистри, дали архитектурата е RISC или CISC, архитектурата и дължината на конвейера, броя на ALU и FPU.*

- Намеренията за използване

  *Зависи за какво се използва компилираната програма. Например тя може да се използва за дебъгиране, за общо ползване (от разнородни потребители на различни машини), за специализирано ползване, или от вградени системи (микроконтролери и др.)*

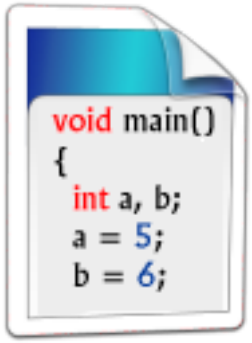
# *Как да влияем на компилатора?*

- Флагове
  *Проблем: те са много (http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Optimize-Options.html).*
- Препроцесор
  *Използвайки директиви като #if, #elseif, #ifdef, #pragma*
- Intrinsics
  *Функции в компилаторът на даден език. Компилаторът познава много добре тези функции и може да генерира по-добър код.*



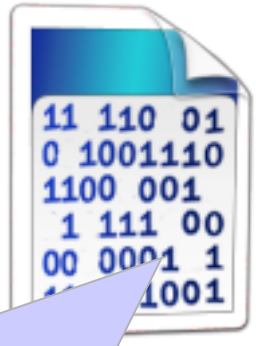
# *Как реагира компилаторът?*

*Език от високо ниво*




Компилатор
(compiler)

*Език от ниско ниво*




```
#include <math.h>
int main() {
    return sin(2.1);
}
```

```
.LFB0:
  .cfi_startproc
  pushq  %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  movq   %rsp, %rbp
  .cfi_def_cfa_register 6
  movabsq  $4606202501814641464, %rax
  movq   %rax, -8(%rbp)
  movl   $0, %eax
  popq   %rbp
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
```

# *Как да изберем/подменим компилатора?*

- Качество/Лекота на използване
  *Компилира ли ми кода лесно, без да се налага много допълнителни настройки?*
- Коректност
  *Получавам ли същите резултати? Как да съм сигурен?*
- Производителност
  *Получавам ли същата производителност? Как да съм сигурен? Има ли нови възможности за повишаване на производителността?*

# *Вие срещу компилатора*

- Очаквате той винаги да прави нещата правилно
- Той очаква вие да правите правилните неща
- Той трябва да е вашият най-добър съюзник
- Не му се доверявайте преди да сте го тествали
- Използвайте повече от един компилатор