Metody Realizacji Języków Programowania

Marcin Benke

MIM UW

3 października 2016

Co to jest kompilator?

Program który tłumaczy programy w języku wyższego poziomu na kod maszynowy procesora (np 80x86, ARM) lub maszyny wirtualnej (np. JVM).

Różnice miedzy interpreterem a kompilatorem:

- interpreter wykonuje program,
- kompilator nie wykonuje programu, a tylko tłumaczy go;
- stworzenie interpretera nie wymaga znajomości maszyny docelowej,
- stworzenie kompilatora wymaga dogłębnej znajomości maszyny docelowej.

Co robi kompilator?

- Wczytuje program, zwykle jako tekst.
- Sprawdza poprawność i dokonuje *analizy* programu.
- “Myśli” chwilę (dokonuje szeregu transformacji programu).
- Generuje kod wynikowy *(synteza)*.

Części kompilatora realizujące analizę i syntezę określa się czasem nazwami *front-end* i *back-end*

Istnieje wiele podobnych klas problemów/programów, gdzie

- analizujemy dane wejściowe (zwykle tekst)
- tłumaczymy na inny “język”.

Analiza

Fazy analizy

- analiza leksykalna — podział na leksemy (“słowa”);
- analiza składniowa — rozbiór struktury programu i jej reprezentacja w postaci drzewa;
- analiza semantyczna — powiązanie użycia identyfikatorów z odpowiednimi deklaracjami; kontrola typów.

Każda z faz analizy powinna dawać czytelne komunikaty o napotkanych błędach

Trudne, ale bardzo ważne.

Faza analizy jest niezależna od języka docelowego.

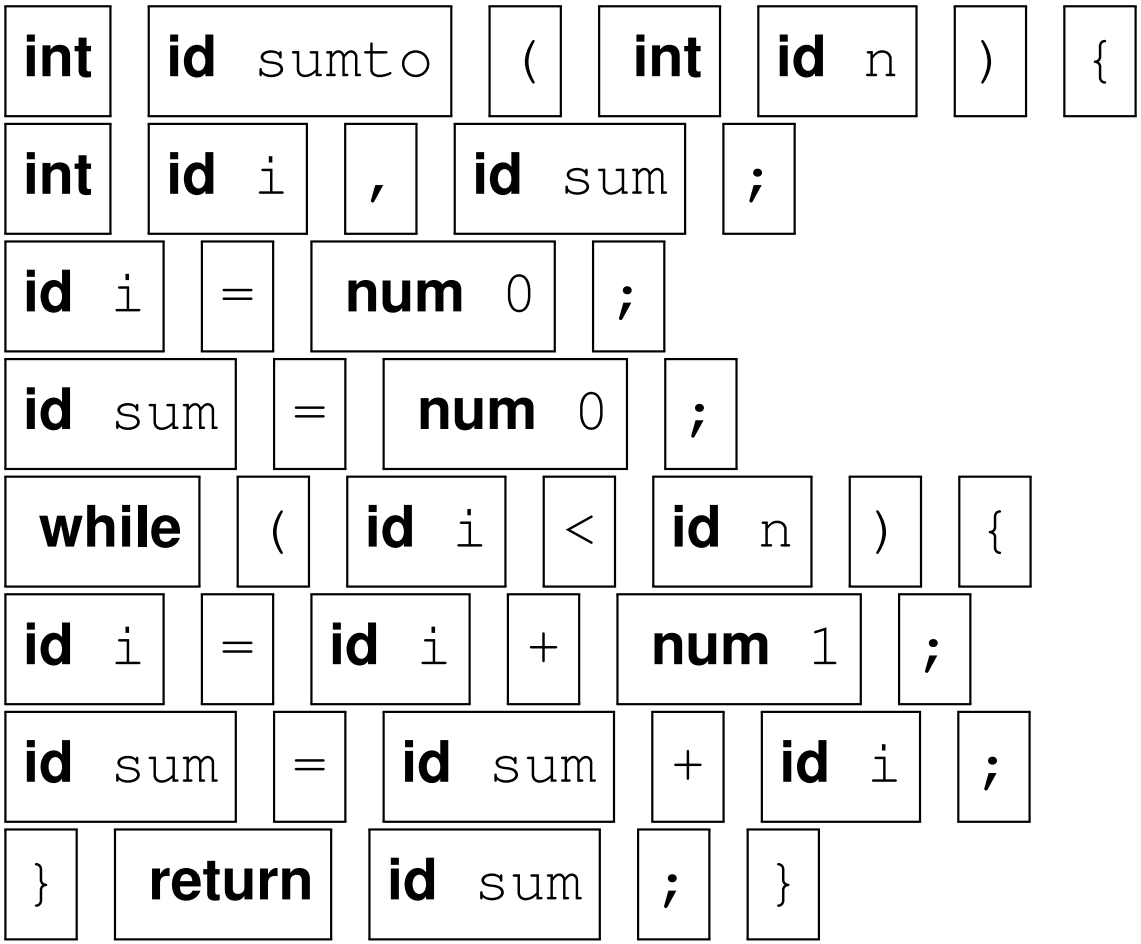
Przykład

Rozważmy prosty program w języku z rodziny C:

```
int sumto(int n)
{
  int i, sum;
  i = 0;
  sum = 0;
  while (i<n) {
    i = i+1;
    sum = sum+i;
 }
 return sum;
}
```

Analiza leksykalna

Dzielimy tekst na *leksemy*:



Analiza składniowa

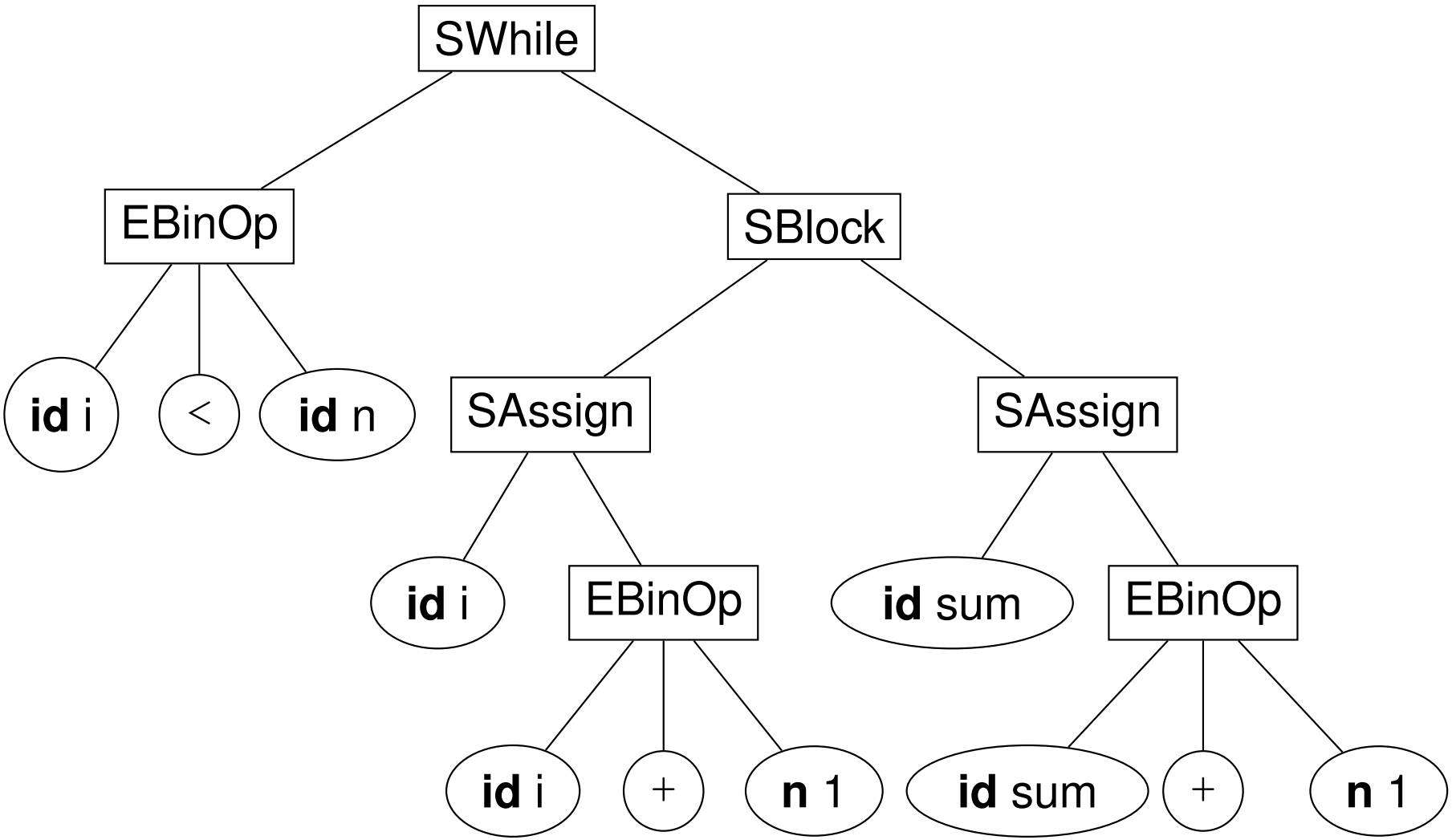
Następnym krokiem jest kontrola poprawności składniowej programu.

Składnię języka opisujemy zwykle przy pomocy gramatyki, np:

$$
\begin{array}{rcl}
\mathit{Stmt} & ::= & \textbf{while}\ (\ \mathit{Exp}\ )\ \mathit{Stmt} \\
& | & \mathit{Var}{=}\mathit{Exp}\ ; \\
& | & \textbf{return}\ \mathit{Exp} \\
& | & \textbf{\{}\ \mathit{Stmts}\ \textbf{\}} \\
\ldots & &
\end{array}
$$

Analiza składniowa

Strumień leksemów zamieniamy na drzewo struktury:



Analiza semantyczna

- Analiza deklaracji
- Zapis informacji w *tablicy symboli*
- Kontrola poprawności użycia symboli i powiązanie z odpowiednimi deklaracjami (poprzez tablicę symboli).
- Kontrola (lub rekonstrukcja) typów.

Maszyny docelowe

- Fizyczna architektura procesora, np x86, x86_64, ARM
- Maszyna wirtualna
  - stosowa, np. JVM
  - rejestrowa, np. LLVM
- Maszyna wirtualna może być użyta jako etap pośredni na drodze do kodu maszynowego
  - *Ahead of Time (AOT)* — generacja kodu maszynowego przed rozpoczęciem wykonania programu (np. LLVM);
  - *Just in Time (JIT)* — generacja kodu w trakcie wykonania, dla wybranych fragmentów programu (np. JVM).

Synteza

- Transformacja drzewa struktury do postaci dogodnej do dalszych przekształceń (kod pośredni)
- Planowanie struktur czasu wykonania (rekordy aktywacji, etc.)
- Ulepszanie kodu (“optymalizacja”)
- Wybór instrukcji
- Szeregowanie instrukcji
- Alokacja rejestrów (dla maszyn rejestrowych)
- Generacja kodu

Maszyna stosowa

Argumenty i wyniki operacji na stosie

(+) Łatwo wygenerować kod z drzewa stuktury, np

```
genIntExp (EBinOp e1 op e2) = do
   genIntExp e1
   genIntExp e2
   intOp op

intOp "+" = emit "iadd"
```

(−) Trudno optymalizować

(−) Realne procesory nie są maszynami stosowymi

`sumto` dla maszyny stosowej (JVM)

```
.method public sumto()I

    iconst_0
    istore_2
    iconst_0
    istore_3

L1: iload_2
    iload_1
    if_icmpge L2
    iload_2
    iconst_1
    iadd
    istore_2
    iload_3
    iload_2
    iadd
    istore_3
    goto  L1

 L2: iload_3
    ireturn
.end method
```

`sumto` w asemblerze 80x86 (gas)

```
        .globl sumto
sumto:
        movl    4(%esp), %ecx ; ecx = n
        xorl    %eax, %eax    ; eax = 0
        testl   %ecx, %ecx    ; ecx <=0?
        jle     .L4           ; skok do L4
        xorl    %edx, %edx    ; edx = 0
.L5:
        addl    $1,%edx       ; edx += 1
        addl    %edx, %eax    ; eax += edx
        cmpl    %edx, %ecx    ; edx != n?
        jne     .L5           ; skok do L5
.L4:
        ret                   ; powrót
```

`sumto` dla LLVM

```
entry:
  %0 = icmp sgt i32 %n, 0
  br i1 %0, label %bb.nph, label %L5
bb.nph:
  %tmp4 = add i32 %n, -2
  %tmp2 = add i32 %n, -1
  %tmp5 = zext i32 %tmp4 to i33
  %tmp3 = zext i32 %tmp2 to i33
  %tmp6 = mul i33 %tmp3, %tmp5
  %tmp7 = lshr i33 %tmp6, 1
  %tmp8 = trunc i33 %tmp7 to i32
  %tmp = shl i32 %n, 1
  %tmp9 = add i32 %tmp, %tmp8
  %tmp10 = add i32 %tmp9, -1
  ret i32 %tmp10
L5:
  ret i32 0
```

Plan wykładu

- Maszyny wirtualne JVM, LLVM
- Analiza semantyczna
- Krótki kurs asemblera x86
- Generacja kodu
- Realizacja funkcji i procedur
- Optymalizacja
- Obsługa wyjątków
- Zarządzanie pamięcią
- Kolokwium (prawdopodobnie 19. grudnia)
- Analiza syntaktyczna
- Kompilacja języków funkcyjnych

Materiały

- `http://moodle.mimuw.edu.pl` Klucz: MRJP1617.3586
- Aho, Lam, Sethi, Ullman; *Compilers: Principles, Techniques, and Tools*, 2/E, Pearson 2006
  (w języku polskim dostępne jest tłumaczenie pierwszego wydania: *Kompilatory. Reguły, metody i narzędzia*, WNT 2002).
- Materiały do wykładu pojawiają się sukcesywnie na **moodle.mimuw.edu.pl**
- Ostateczna wersja notatek **po wykładzie**.
- Kontakt ze mną: **ben@mimuw.edu.pl**
- Konsultacje: poniedziałki 1400-1530 pokój 5750, proszę się umawiać.

Sprawy organizacyjne

Wykład+ćwiczenia
Kolokwium (koniec grudnia lub na początku stycznia). na wykładzie (kolokwium poprawkowe pod koniec stycznia).

Laboratorium: piszemy kompilator dla prostego języka

- Etap 1: wyrażenia arytmetyczne+przypisanie — kod JVM+LLVM
- Etap 2: Latte — kod LLVM albo x86
- Etap 3: opcjonalnie na wyższą ocenę — rozszerzenia języka, etc.
- Projekt oddawany etapami, oddawanie w terminie bardzo ważne.

Obecność (i aktywność!) na zajęciach jest wskazana.
Prowadzący mają prawo (a nawet obowiązek) skreślić osoby, które opuszczą bez usprawiedliwienia zbyt wiele zajęć.

Proszę zadawać pytania!

Zasady zaliczania

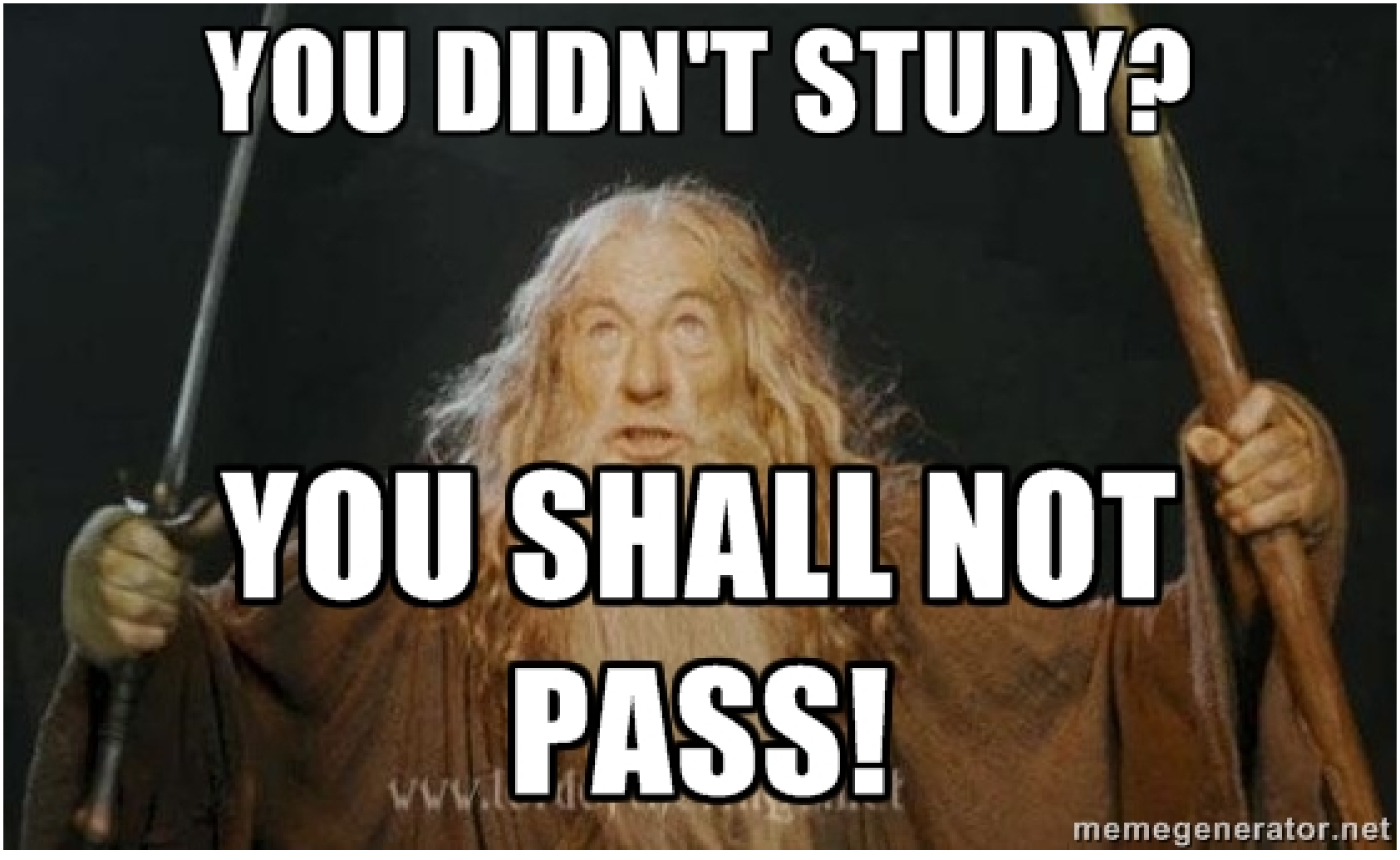
Egzamin 55%, projekt zaliczeniowy 30% kolokwium 15%.

Dla zaliczenia trzeba oddać wszystkie programy i uzyskać z nich min 50% punktów oraz min. 50% z kolokwium.

Zaliczenie jest wymagane do przystąpienia do egzaminu w pierwszym terminie.

Do przystapienia do egzaminu w drugim terminie, wymagane jest uzyskanie przynajmniej 30% możliwych do uzyskania punktów za projekt;

Punkty uzyskane za kolokwium i projekt zaliczeniowy są wliczane do oceny końcowej także w drugim terminie.


YOU DIDN'T STUDY?
YOU SHALL NOT PASS!
memegenerator.net

Maszyna wirtualna Javy

1. Maszyna abstrakcyjna
   - izoluje od problemów konkretnej architektury
   - standaryzowany opis
   - wiele implementacji,
2. Wykonywanie programów niskopoziomowych
3. Zapewnia przenośność
4. Zapewnia pewne mechanizmy bezpieczenstwa

Bajtkod

JVM, jak większość maszyn, wykonuje kod binarny, tzw. bajtkod

Pliki `class` zawierają bajtkod plus dodatkowe informacje

Kompilatory Javy (np. `javac`) generują pliki `class`

Bajtkod ma też swoją reprezentację tekstową,
można ją zobaczyć np. przy pomocy `javap`

Jasmin tłumaczy reprezentację tekstową na bajtkod

Istnieją też biblioteki do bezpośredniej generacji bajtkodu
np. Apache Commons BCEL

Jasmin

- Java ASM
- tłumaczy tekstowy opis kodu JVM na plik .class
- `java -jar jasmin.jar hello.j`
- `http://jasmin.sourceforge.net/`

Maszyna wirtualna Javy

Typy danych

- Typy bazowe: całkowite (`int`, etc.), zmiennoprzecinkowe (`float`, `double`)
- Referencje do obiektów

Obszary danych

- Zmienne lokalne i parametry są przechowywane na stosie.
- Stos służy też do obliczeń.
- Obiekty (w tym tablice) przechowywane na stercie.
- Stałe zmiennoprzecinkowe i napisowe przechowywane w obszarze stałych — nie musimy się tym przejmować jeśli używamy Jasmina.

Stos JVM

- Stos jest ciągiem *ramek*. Każda instancja metody ma swoją ramkę.
- Różne postaci wywołania:
  - `invokestatic` dla metod statycznych (np. dla funkcji Latte)
  - `invokevirtual` dla metod obiektowych
  - `invokespecial` np. dla konstruktorów (dawniej `invokenonvirtual`)
- JVM zajmuje się kwestiami porządkowymi, jak:
  - alokacja i zwalnianie ramek
  - przekazywanie parametrów
  - dostarczanie wyników

Struktura ramki stosu

Ramka zawiera zmienne lokalne (w tym parametry) i stos operandów (dla obliczeń). Rozmiary tych obszarów muszą być znane podczas kompilacji.

Obszar zmiennych lokalnych

Tablica słów przechowująca argumenty i zmienne lokalne

- `double` zajmują po dwa słowa, `int`, referencje — jedno.
- Dla metod instancyjnych pod indeksem 0 jest `this`, dla statycznych — pierwszy argument.

Stos operandów

- Element mieści wartość dowolnego typu.
- Przed wywołaniem kładziemy argumenty na stosie, po powrocie wynik tamże.

Przykład programu – prosta klasa

```
class Simple {
  public static void main(String argv[]) {
    for (int i=1;i<=10;i++){
      System.out.println(i);
    }
  }
}

$ javac Simple.java
$ javap -c Simple
```

Przykład programu – prosta klasa

```
Method Simple()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return
Method void main(java.lang.String[])
   0 iconst_1
   1 istore_1
   2 goto 15
   5 getstatic #2 <Field java.io.PrintStream out>
   8 iload_1
   9 invokevirtual #3 <Method void println(int)>
  12 iinc 1 1
  15 iload_1
  16 bipush 10
  18 if_icmple 5
  21 return
```

Instrukcje JVM

Maszyna stosowa

`load` n	załaduj n-tą zmienną lokalną (także parametry)
`store` n	zapisz wartość ze stosu do zmiennej lokalnej
`push` val	wstaw stałą na stos
`add, sub, mul,...`	operacje arytmetyczne
`ldc` stała	załaduj stałą z tablicy stałych
`getfield` vname cname	pobierz pole z obiektu
`getstatic` vname cname	pobierz pole z klasy
`putfield` vname cname	ustaw pole obiektu

Instrukcje takie jak `load, store,add` są prefiksowane typami, zatem np. `aload, istore, fadd,...`

Instrukcje JVM (2)

`checkcast` c	sprawdź czy obiekt jest danej klasy
`invokeVirtual` m	wywołanie metody
`invokeSpecial` m	wywołanie inicjalizatora, metody prywatnej etc.
`tReturn`	powrót (prefiksowane typem)
`pop`	zdejmij ze stosu
`goto` adres	skok bezwarunkowy
`if_icmpGe` adres	weź ze stosu b, a , skocz gdy $a \geqslant b$; także: eq, ne, lt, gt,le
`ifEq` adres	weź ze stosu a (int), skocz gdy $a = 0$; także: ne, lt, gt,le, ge

Kompilacja — przekazywanie argumentów

```
int addTwo(int i, int j) {
    return i + j;
}
```

kompiluje się do:

```
Method int addTwo(int,int)
   iload_1   // zmienna lokalna 1 na stos (i)
   iload_2   // zmienna lokalna 2 na stos (j)
   iadd      // dodaj; wynik na szczycie stosu
   ireturn   // powrót z wynikiem
```

`this` w zmiennej lokalnej 0

Kompilacja – wywoływanie metod

```
int add12and13() {
    return addTwo(12, 13);
}
```

kompiluje się do:

```
Method int add12and13()
  aload_0             // this na stos
  bipush 12           // 12 na stos
  bipush 13           // 13 na stos
  invokevirtual addtwo(II)I // wywołanie metody
  ireturn             // powrót z wynikiem, to jest
                      // tez wynik addTwo()
```

Jasmin — hello

```
.class  public Hello
.super  java/lang/Object

; standard initializer
.method public <init>()V
   aload_0
   invokespecial java/lang/Object/<init>()V
   return
.end method
```

Jasmin — hello c.d.

```
.method public static main([Ljava/lang/String;)V
.limit stack 2
  getstatic
   java/lang/System/out Ljava/io/PrintStream;
  ldc "Hello"
  invokevirtual
   java/io/PrintStream/println(Ljava/lang/String;)V
  return
.end method
```

Deskryptory metod

Deskryptory metod są postaci

```
pakiet/Klasa/metoda(typy argumentów)typ wyniku
```

niektóre typy

`V`	void
`I`	integer
`Z`	boolean
`Lcname;`	klasa `cname`
`[t`	tablica elementów typu `t`

Przykłady:

```
addTwo(II)I
main([Ljava/lang/String;)V
java/io/PrintStream/println(Ljava/lang/String;)V
```

Java – sumto

```
public class Sumto {
  int sumto(int n)
  {
    int i, sum;
    i = 0;
    sum = 0;
    while (i<n) {
      i = i+1;
      sum = sum+i;
    }
    return sum;
  }
}
```

Jasmin — sumto

```
.method public sumto(I)I
.limit locals 4
.limit stack 2
```

```
    iconst_0
    istore_2
    iconst_0
    istore_3
```

```
L1: iload_2
    iload_1
    if_icmpge L2
    iload_2
    iconst_1
    iadd
    istore_2
    iload_3
    iload_2
    iadd
    istore_3
    goto  L1
```

```
 L2: iload_3
     ireturn
.end method
```

JVM — tablice

Instrukcje

- `newarray typ` — utwórz tablicę (rozmiar na stosie)
- `iaload` załaduj element tablicy `int` (tablica i indeks na stosie)
- `aastore` zapisz do tablicy referencji (tablica, indeks, wartość na stosie)

Przykład

```
public class Arr {
  public static void main(String argv[]){
    int[] a = new int[3];
    a[2] = 42;
    System.out.println(argv[1]);
  }
}
```

Tablice - kod JVM (istotny fragment)

```
; int[] a = new int[3];
iconst_3
newarray int
astore_1
; a[2] = 42;
aload_1
iconst_2
bipush 42
iastore
; System.out.println(argv[1]);
getstatic java/lang/System/out
          Ljava/io/PrintStream;
aload_0
iconst_1
aaload
invokevirtual
 java/io/PrintStream/println(Ljava/lang/String;)V
```

JVM — obiekty/atrybuty

Dyrektywa `.field typ`

Instrukcje

- `new typ`
- `getfield klasa/pole typ`
- `putfield klasa/pole typ`

```
public class Lista {
  int car;
  Lista cdr;
static int cadr(Lista a) {
  return a.cdr.car;
}
public static void main(String args[]) {
    Lista l = new Lista();
    l.car = 42;
    l.cdr = new Lista();
    System.out.println(cadr(l));
}}
```

JVM — obiekty/atrybuty

Kod JVM (interesujący fragment)

```
.field car I
.field cdr LLista;
.method static cadr(LLista;)I
 aload_0
 getfield Lista/cdr LLista;
 getfield Lista/car I
 ireturn
.end method
```

JVM — obiekty/atrybuty

```
.method public static main([Ljava/lang/String;)V
  new Lista
  dup
  invokespecial Lista/<init>()V
  astore_1
  aload_1
  bipush 42
  putfield Lista/car I
  aload_1
  new Lista
  dup
  invokespecial Lista/<init>()V
  putfield Lista/cdr LLista;
  invokestatic Lista/cadr(LLista;)I
  invokevirtual java/io/PrintStream/println(I)V
  return
```