Compilatoare

Compilare Just-In-Time
Limbaje dinamice

CPL IR crunch

Interpretare vs. compilare

- Interpretarea – ideala pentru dezvoltarea de aplicatii distribuite
  - Limbaje dinamice flexibile (pot adauga tipuri de date si functii la runtime)
  - Programe portabile pe mai multe arhitecturi
    - In practica: "Write once, ~~run~~ debug anywhere"
  - Securitate – posibilitatea de a rula codul intr-un "sandbox".
- Dezavantajul major – performanta (10x mai lent)
- Solutii
  - Virtual Machines (JVM, CLR, Dalvik)
    - Reduce nivelul codului interpretat, dar introduce un pas de compilare.
  - Compilarea Just-In-Time (JVM, Dalvik)
  - Compilarea Ahead-Of-Time – la instalarea aplicatiei (ART)

CPL IR crunch

Just-In-Time

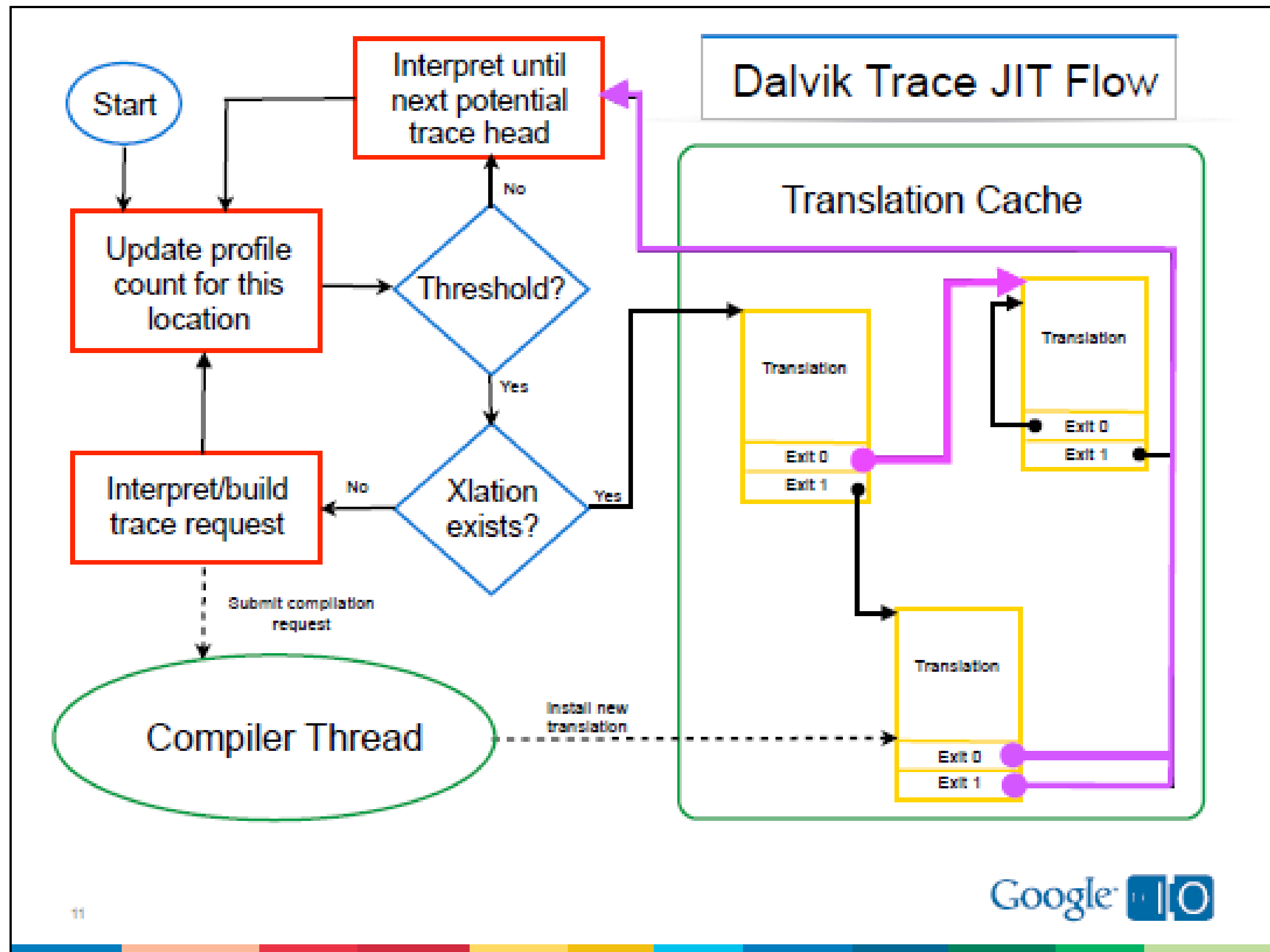
- Compilatorul ruleaza la momentul executiei programului.
- Trebuie minimizata **suma** timpului de compilare si executie
- Cand compilam:
  - La instalarea aplicatiei, la lansare, la executia unei rutine, la executia unei instructiuni
- Ce compilam:
  - Un program, o biblioteca, o pagina de memorie, o metoda, o secventa (“trace”), o instructiune

CPL IR crunch

Cand compilam

- Compilare continua
  - Compilatorul ruleaza in background pe un thread separat de interpretor
  - Rezultatul - functii native pe care interpretorul le poate incarca dinamic si le poate folosi (de ex. pentru a interpreta un apel de metoda)
- Smart Just-In-Time
  - Profiling - estimeaza castigul rezultat din compilare vs timpul de compilare.
  - Daca e pozitiv, opreste interpretarea si porneste compilarea.
  - Posibil mai multe nivele de optimizare (e.g. Mozilla Baseline / IonMonkey compilers)

Exemplu de integrare : Dalvik



CPL IR crunch

Generare de cod

- Exemplu: JVM
  - Fiecare stack frame contine variabile locale si o stiva de operanzi.
  - Tipul operanzilor de pe stiva este cunoscut la compilare
- Tipuri de instructiuni:
  - Load/store intre locale si stiva de operanzi
  - Aritmetica cu operanzi pe stiva
  - Crearea de obiecte si apeluri de metoda
  - Acces la campuri si elemente de array
  - Salturi si exceptii

JVM – generare de cod

- Exemplu:

```
iconst 2
iload a
iload b
iadd
imul
istore c
```

a	42
b	7
c	0

Stack
7
42
2

- Calculeaza: `c := 2 * (a + b)`

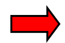
Lazy Code Selection

- Simuleaza static executia instructiunilor JVM de pe stiva de operanzi
- Genereaza instructiuni native in timpul simularii.
- Stiva simulate contine *locatia* si nu *valoarea* unui operand

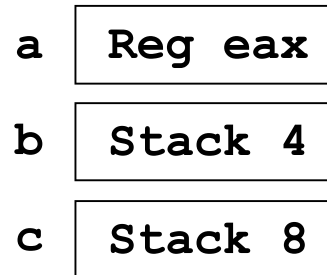
Unde: Intel Vtune JIT compiler ([Adl-Tabatabai 98])

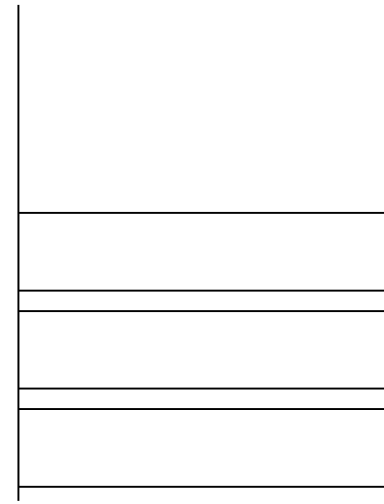
JVM – Lazy Code Selection

- Exemplu:



```
iconst 2
iload a
iload b
iadd
imul
istore c
```

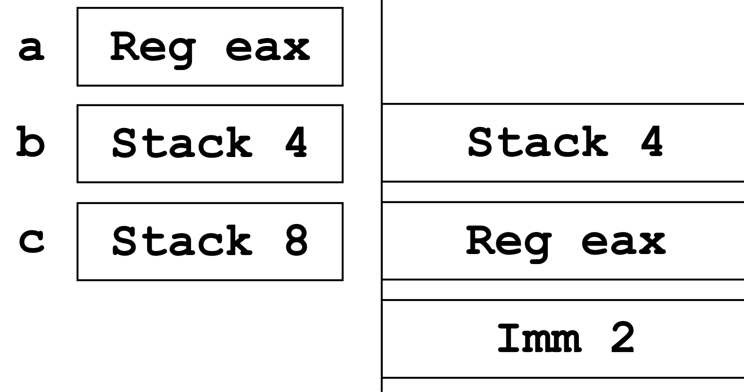




JVM – Lazy Code Selection

- Exemplu:

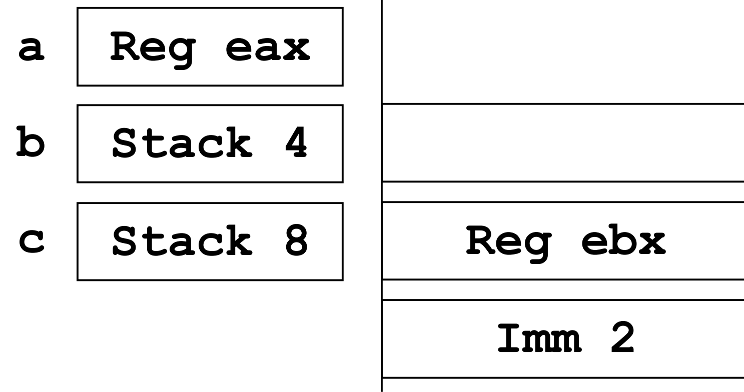
```
iconst 2
iload a
iload b
iadd
imul
istore c
```



JVM – Lazy Code Selection

- Exemplu:

```
iconst 2
iload a
iload b
iadd
imul
istore c
```

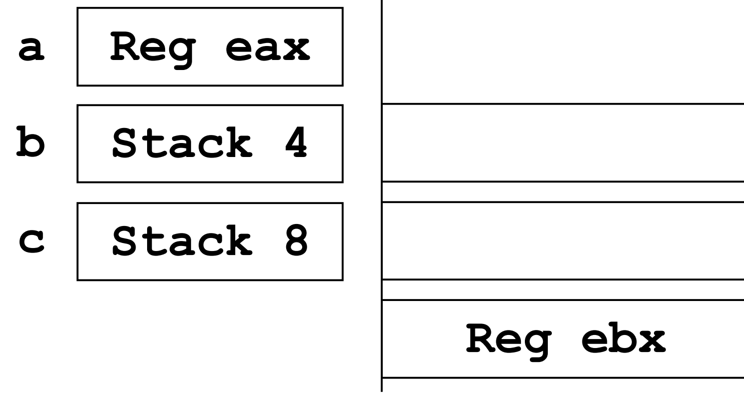



```
movl ebx, eax
addl ebx, 4(esp)
```

JVM – Lazy Code Selection

- Exemplu:

```
iconst 2
iload a
iload b
iadd
imul
istore c
```

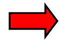


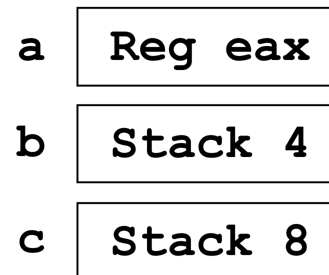

```
movl ebx, eax
addl ebx, 4(esp)
sall ebx, 1
```

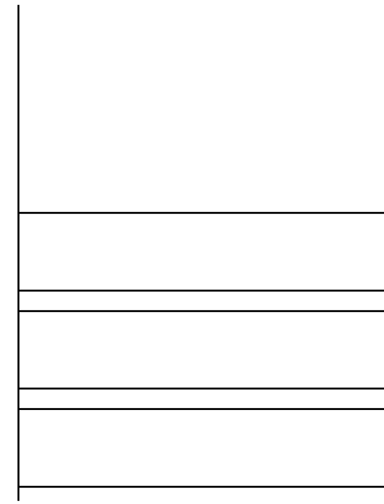
JVM – Lazy Code Selection

- Exemplu:

```
iconst 2
iload a
iload b
iadd
imul
istore c
```







```
movl ebx, eax
addl ebx, 4(esp)
sall ebx, 1
movl 8(esp), ebx
```

JVM - generare de cod

- Converteste cod pt masina stiva → registri
- Selecteaza instructiuni x86 complexe
- Strength reduction, constant propagation, method inlining
- Alocare de registri simpla dar rapida (insa spill la sfarsitul unui basic block)
- Versiuni simple ale optimizarilor clasice
  - "Pretty good is good enough"
  - Slow down: 2x JIT vs 10x interpretare

Limbaje dinamice

- O istorie lunga… Perl, Javascript, Python, Ruby, Tcl
- Probleme
  - Lente, greu de optimizat
  - Greu de creat tool-uri avansate (IDE cu navigare, refactoring, analiza statica)
  - Scalarea: Dezvoltarea e rapida initial, dar din ce in ce mai dificila atunci cand aplicatiile cresc.

CPL IR crunch

De ce sunt lente

- Interpretoarele sunt gandite pentru scripting (I/O bound)
- Compilarea traditionala (statica) nu genereza cod eficient
  - Tipul variabilelor se poate schimba
  - Layoutul obiectelor se poate schimba
  - Obiectele pot primi metode noi create la runtime
- Exemplu: un apel de functie/metoda:
  - In C: o instructiune “call”
  - In C++: instructiuni multiple: lookup in vtable + call
  - In JavaScript: lookup-ul unui sir intr-un dictionar

CPL IR crunch

Javascript

- Sintaxa stil “Java”, dar OOP bazat pe prototipuri, nu clase!
  - Single dispatch (similar cu C++, Java)
    - Alegerea metodei e bazata pe tipul lui “this”, nu si pe cel al celorlalte argumente.
  - Fiecare obiect are propria “clasa”.
  - Codul unei functii e o valoare. Suporta closures.

- Limbajul standard pentru aplicatii (web si nu numai), cel mai portabil → focus pe performanta.
- Poate fi limbaj target pentru un compilator “clasic” (Emscripten / asm.js)

CPL IR crunch

Compilarea eficienta JIT

- Decizii la runtime, nu statice, bazate pe euristici
  - Profiling → informatii “probabilistice” despre tipul si valorile variabilelor
  - Optimizari simple si rapide bazate pe tipurile si valorile cele mai probabile ale obiectelor.
  - Apeluri de metode→ “Polymorphic inline cache”

CPL IR crunch

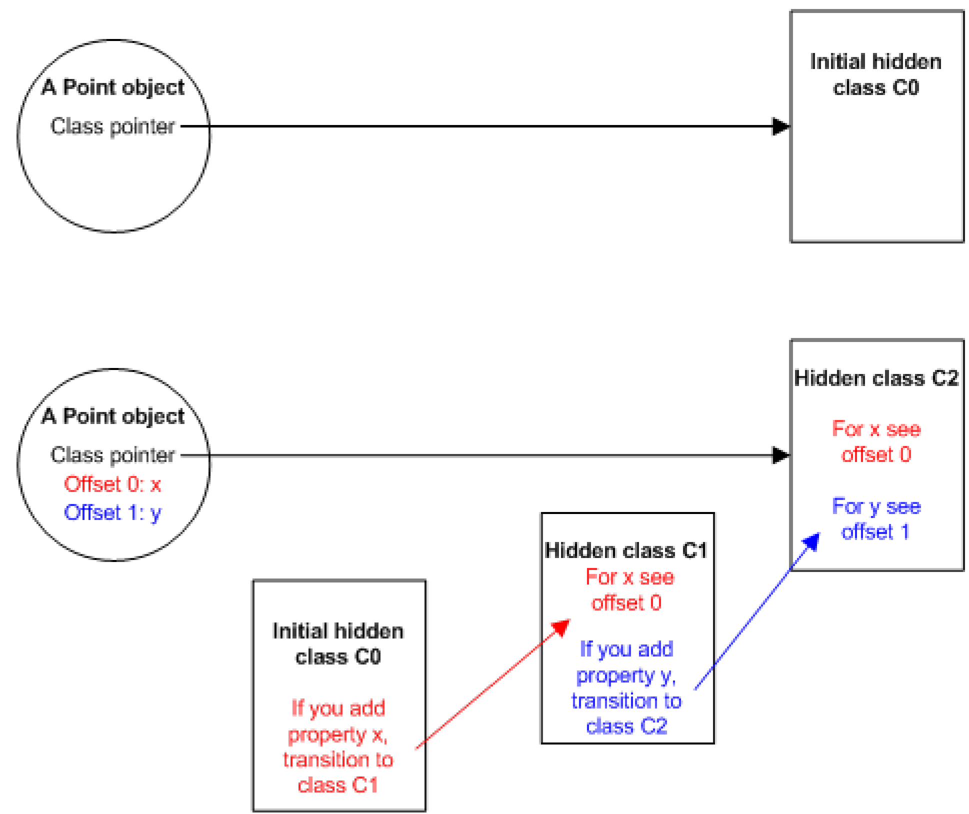
Layoutul obiectelor

- C++, Java – layoutul e cunoscut static
  - O singura instructine “load” necesara
- Javascript – layoutul se schimba dynamic
  - Adresa unui camp se obtine printr-un lookup in hash-table bazat pe sir.
  - In realitate, schimbarile de layout nu sunt frecvente in cod.
  - Obiecte diferite au de obicei layout-uri similare.
- Solutie: “hidden classes”

Layoutul obiectelor (Google V8)

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

new Point(x, y);
```



https://developers.google.com/v8/design

Exemplu: Google V8

- Interpretorul parcurge AST-ul, genereaza cod nativ (JIT), il apeleaza.
- In cazul in care tipul obiectului se schimba, se sare inapoi in codul interpretorului.

```
point.x
```

```
# ebx = the point object
cmp [ebx,<hidden class offset>],<cached hidden class>
jne <inline cache miss>
mov eax,[ebx, <cached x offset>]
```

Apelurile de metoda

"Inline caching"

- Obj.ToString() – trebuie cunoscut tipul lui "Obj" pentru a apela "ToString"
- Statistici pentru fiecare instructiune de call.
- La primul apel: se cauta metoda in dictionar.
- La apelurile ulterioare: se verifica tipul obiectului, apoi se apeleaza metoda, daca nu s-a schimbat.
- "Polymorphic inline cache" – se pastreaza adresele de metode pentru 2 sau mai multe tipuri.

```
var values = [1, "a", 2, "b", 3, "c", 4, "d"];
for (var value in values) { document.write(value.toString()); }
```

CPL
IR crunch

Trace compiling

- Reprezentarea interna: “trace tree” in loc de CFG.
- Profiling: stabileste caile cele mai folosite in program, generaza cod pentru un trace, nu pentru o metoda.
- Optimizari implicite: inlining, inferenta de tipuri, alocare simpla de registri, forma apropiata SSA.
- Overhead mic pentru bucle

Folosit: Adobe Tamarin, Firefox (older engines), Dalvik

CFG traditional

```
1:  code;
2:  do {
        if (condition) {
3:          code;
        } else {
4:          code;
        }
5:  } while (condition);
6:  code;
```

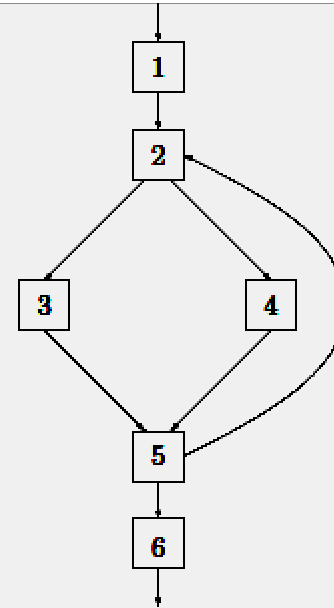


Figure 1. A sample program and its corresponding control flow graph. The starting point of each basic block is indicated through a corresponding label in the program code.

Trace tree

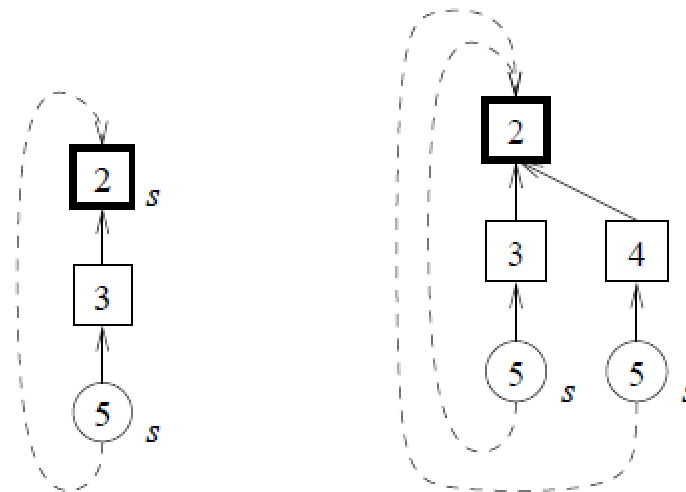


Figure 3. The left figure shows an initial trace recorded for the loop construct shown in Figure 1. Side exits nodes are labeled with *s*. The right figure shows lazy extension of the trace tree after executing the initial trace $(2, 3, 5)$ and encountering a side exit at node 2. The new trace shares all instructions between the anchor node and the side exit with the previous trace it branched off from, which in this example is only the anchor node 2 itself.

CPL IR crunch

Trace tree

- Construit la runtime
- Ideal pentru bucle interioare executate des
- Punctul de inceput (loop header) detectat din informatii de profile.
- Codul este compilat pentru tipul cel mai probabil al variabilelor.
- Din Trace tree se iese inapoi in interpretor.

CPL IR crunch

Bucle in trace trees

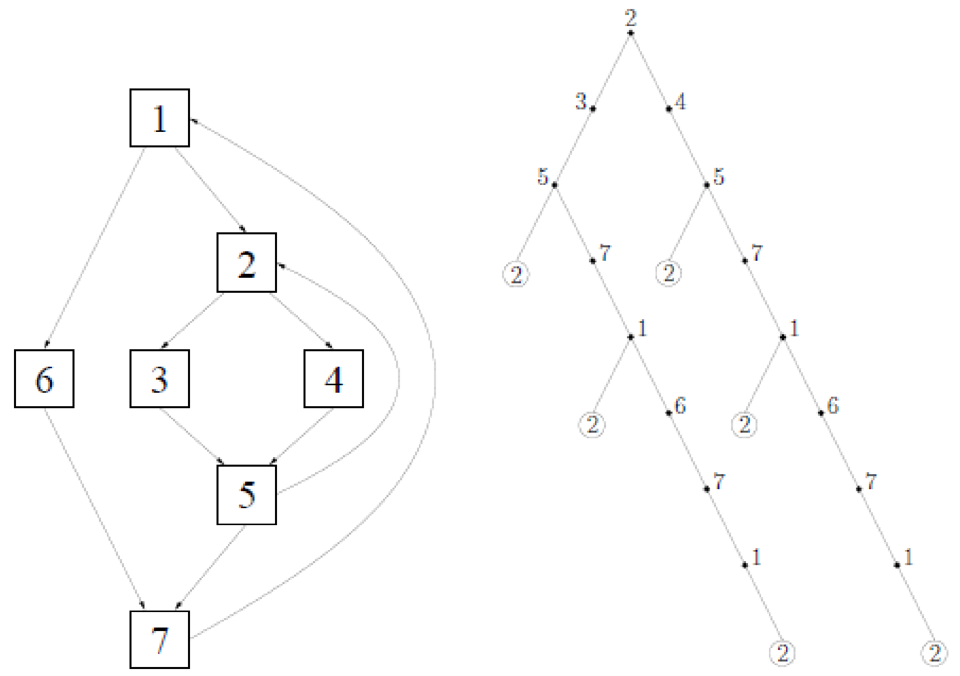


Figure 4. Example of a nested loop and a set of suitable traces through the nested loop. For simplicity, implicit edges are shown as a simple edge going to a copy of the anchor node 2, instead of a dashed edge back to the actual anchor node 2.

Tools

- IDE-ul modern: autocomplete, jump-to-definition, browsing, refactoring
- Tehnici similare cu DFA!
- Informatii despre tipuri:
  - Din sintaxa

```
var x = 12.5;
var y = { a:1, b:2};
function foo(a, b) { return a+b;}
```

  - Din inferenta de tipuri

```
var x = 12.5; var y=x;
```

  - Din coding style / code best practices / jsdoc comments