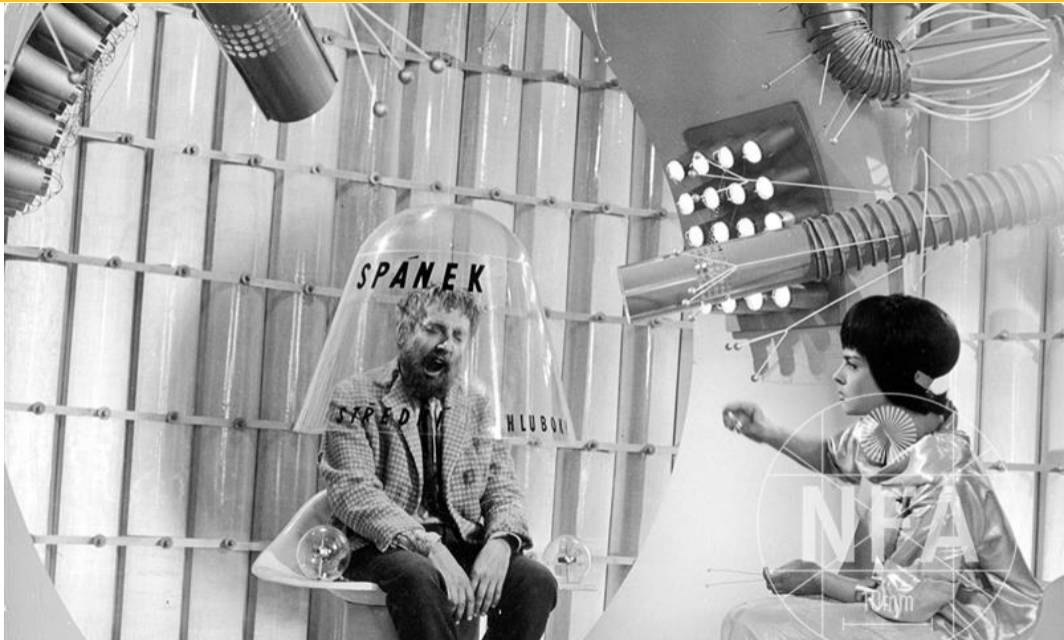


Překladače (3. přednáška)

Jan Hubička

Katedra aplikované matematiky
Univerzita Karlova
Praha

23. března 2020





Minulé přednášky

- Open-source překladače: GCC, LLVM, Open64
- Just in time překladače: (SpiderMonkey, V8)

Minulé přednášky

- Open-source překladače: GCC, LLVM, Open64
- Just in time překladače: (SpiderMonkey, V8)
- Základní organizace překladače:
 - Při překladače:
 - preprocesor
 - překladač (lexikální analýze, syntaktická analýzy – parser, generování mezikódu a jednotlivé optimalizace)
 - assembler
 - linker,

Minulé přednášky

- Open-source překladače: GCC, LLVM, Open64
- Just in time překladače: (SpiderMonkey, V8)
- Základní organizace překladu:
 - Při překladu:
 - preprocesor
 - překladač (lexikální analýze, syntaktická analýzy – parser, generování mezikódu a jednotlivé optimalizace)
 - assembler
 - linker,
 - Při instalaci:
 - reoptimalizace
 - prelinking

Minulé přednášky

- Open-source překladače: GCC, LLVM, Open64
- Just in time překladače: (SpiderMonkey, V8)
- Základní organizace překladu:
 - Při překladu:
 - preprocesor
 - překladač (lexikání analýze, syntaktická analýzy – parser, generování mezikódu a jednotlivé optimalizace)
 - assembler
 - linker,
 - Při instalaci:
 - reoptimalizace
 - prelinking
 - Při běhu:
 - dynamický linker
 - JIT překladač

Minulé přednášky

- Open-source překladače: GCC, LLVM, Open64
- Just in time překladače: (SpiderMonkey, V8)
- Základní organizace překladu:
 - Při překladu:
 - preprocesor
 - překladač (lexikální analýze, syntaktická analýzy – parser, generování mezikódu a jednotlivé optimalizace)
 - assembler
 - linker,
 - Při instalaci:
 - reoptimalizace
 - prelinking
 - Při běhu:
 - dynamický linker
 - JIT překladač

Minulé přednášky

- Open-source překladače: GCC, LLVM, Open64
- Just in time překladače: (SpiderMonkey, V8)
- Základní organizace překladače:
 - Při překladu:
 - preprocesor
 - překladač (lexikání analýze, syntaktická analýzy – parser, generování mezikódu a jednotlivé optimalizace)
 - assembler
 - linker,
 - Při instalaci:
 - reoptimalizace
 - prelinking
 - Při běhu:
 - dynamický linker
 - JIT překladač
- Základní vlastnosti mezijazyka
 - tabulka symbolů ←
 - typy
 - instrukce/statementy – aritmetika
 - práce s pamětí
 - řízení toku

GIMPLE — high-level mezijazyk GCC

`gcc -fdump-tree-all` vypíše GIMPLE po jednotlivých průchodech

Example

GIMPLE — high-level mezijazyk GCC

`gcc -fdump-tree-all` vypíše GIMPLE po jednotlivých průchodech

Example

```
C:  
int fact (int a)  
{  
    int ret = 1;  
    for (;a>1;a--)  
        ret *= a;  
    return ret;  
}
```

GIMPLE — high-level mezijazyk GCC

gcc -fdump-tree-all vypíše GIMPLE po jednotlivých průchodech

Example

C:

```
int fact (int a)
{
  int ret = 1;
  for (; a > 1; a--)
    ret *= a;
  return ret;
}
```

GIMPLE:

```
fact (int a)
{
  int D.1918;
  int ret;
  ret = 1;
  goto <D.1915>;
<D.1914>:
  ret = ret * a;
  a = a + -1;
  goto <D.1915>;
if (a > 1) goto <D.1914>; else goto <D.1916>;
<D.1916>:
  D.1918 = ret;
  return D.1918;
}
```

ASSIGNMENT

RTL — low level mezijazyk GCC

gcc -fdump-rtl-all-slim vypíše RTL po jednotlivých průchodech (v úsporné syntaxi)

Example

C: **SINGLE INT** **INTEGER** **CISLO INSTR**

```

int fact (int a)
{
  int ret = 1;
  for (; a > 1; a--)
    ret *= a;
  return ret;
}

```

RTL: **POZNÁMKA**

```

6: NOTE_INSN_BASIC_BLOCK 2
38: r85:SI=di:SI
2: r83:SI=r85:SI
3: NOTE_INSN_FUNCTION_BEG
10: flags:CCGC=cmp(r83:SI,0x1)
11: pc={ (flags:CCGC<=0)?L31:pc}
26: NOTE_INSN_BASIC_BLOCK 3
4: r82:SI=0x1
15: L15:
12: NOTE_INSN_BASIC_BLOCK 4
13: {r82:SI=r82:SI+r83:SI;clobber flags:CC;}
14: {r83:SI=r83:SI-0x1;clobber flags:CC;}
16: flags:CCZ=cmp(r83:SI,0x1)
17: pc={ (flags:CCZ!=0)?L15:pc}
31: L31:
30: NOTE_INSN_BASIC_BLOCK 5
5: r82:SI=0x1
22: L22:
25: NOTE_INSN_BASIC_BLOCK 6
23: ax:SI=r82:SI
REG_DEAD r82:SI
24: use ax:SI

```

PSEUDO REGISTER → r85:SI, r83:SI, r82:SI

HARWARE REGISTER → r85:SI, r83:SI, r82:SI

PROG COUNTER → pc={ (flags:CCGC<=0)?L31:pc}

ADDMIVKA → ret *= a;

NAZOBELI → a--

DEKR. → r82:SI=0x1

RTL — low level mezijazyk GCC

Na konci kompilace je téměř 1:1 souvislost mezi instrukcema RTL a instrukcema assembleru.

Example

Finální RTL:

```
6: NOTE_INSN_BASIC_BLOCK 2
4: ax:SI=0x1
10: flags:CCGC=cmp(di:SI,0x1)
11: pc={ (flags:CCGC<=0)?L31:pc}
15: L15:
12: NOTE_INSN_BASIC_BLOCK 3
13: {ax:SI=ax:SI+di:SI;clobber flags:CC;}
14: {di:SI=di:SI-0x1;clobber flags:CC;}
16: flags:CCZ=cmp(di:SI,0x1)
17: pc={ (flags:CCZ!=0)?L15:pc}
39: NOTE_INSN_BASIC_BLOCK 4
47: use ax:SI
40: simple_return
41: barrier
31: L31:
30: NOTE_INSN_BASIC_BLOCK 5
24: use ax:SI
45: simple_return
46: barrier
```

assembler:

```
.file "bbs.c"
.text
.p2align 4
.globl fact
.type fact, @function

fact:
.LFB0:
.cfi_startproc
movl $1, %eax # 4
cmpl $1, %edi # 10
jle .L4 # 11
.p2align 4,,10
.p2align 3
.L3:
imull %edi, %eax # 13
subl $1, %edi # 14
cmpl $1, %edi # 16
jne .L3 # 17
ret # 40
.p2align 4,,10
.p2align 3
.L4:
ret # 45
.cfi_endproc
```

PODHLKA {

TĚLO
SKLOKAS }

LLVM

clang -mllvm -print-after-all vypíše RTL po jednotlivých průchodech (v úsporné syntaxi)

Example

C:

```
int fact (int a)
{
    int ret = 1;
    for (;a>1;a--)
        ret *= a;
    return ret;
}
```

LLVM:

```
; ModuleID = 'bbs.c'
source_filename = "bbs.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: nounwind uwtable
define dso_local i32 @fact(i32) #0 {
    br label %2

2:
    %3 = phi i32 [ %0, %1 ], [ %8, %6 ]
    %4 = phi i32 [ 1, %1 ], [ %7, %6 ]
    %5 = icmp sgt i32 %3, 1
    br i1 %5, label %6, label %9

6:
    %7 = mul nsw i32 %4, %3
    %8 = add nsw i32 %3, -1
    br label %2

9:
    ret i32 %4
}
```

Annotation: "POZDESI" with a red arrow pointing to the entry block (2:).

Red annotations in LLVM code:

- Red arrow pointing to the entry block (2:).
- Red box around the entry block (2:).
- Red arrow pointing to the phi nodes.
- Red arrow pointing to the icmp instruction.
- Red arrow pointing to the br instruction.
- Red arrow pointing to the phi nodes in block 6:.
- Red arrow pointing to the mul instruction in block 6:.
- Red arrow pointing to the add instruction in block 6:.
- Red arrow pointing to the br instruction in block 6:.
- Red box around the exit block (9:).

LLVM

clang -mllvm -print-after-all vypíše RTL po jednotlivých průchodech (v úsporné syntaxi)

Example

LLVM IR:

Function Live Ins: \$edi in %5

bb.0 (%ir-block.1):

successors: %bb.1(0x40000000), %bb.3(0x40000000); %bb.1(50.00%), %bb.3(50.00%)

liveins: \$edi

%5:gr32 = COPY \$edi

%6:gr32 = MOV32ri 1

CMP32ri8 %5:gr32, 2, implicit-def \$eflags

JCC_1 %bb.3, 12, implicit \$eflags

JMP_1 %bb.1

bb.1 (%ir-block.3):

; predecessors: %bb.0

successors: %bb.2(0x80000000); %bb.2(100.00%)

%8:gr32 = MOV32ri 1

bb.2 (%ir-block.4):

; predecessors: %bb.1, %bb.2

successors: %bb.2(0x7c000000), %bb.3(0x04000000); %bb.2(96.88%), %bb.3(3.12%)

%0:gr32 = PHI %8:gr32, %bb.1, %2:gr32, %bb.2

%1:gr32 = PHI %5:gr32, %bb.1, %3:gr32, %bb.2

%2:gr32 = nsw IMUL32rr %0:gr32(tied-def 0), %1:gr32, implicit-def dead \$eflags

%3:gr32 = nsw ADD32ri8 %1:gr32(tied-def 0), -1, implicit-def dead \$eflags

CMP32ri8 %3:gr32, 1, implicit-def \$eflags

JCC_1 %bb.2, 15, implicit \$eflags

Control Flow Graph (CFG)

INTRA PROCEDURNÍ OPT

Definition (Control Flow Graph (CFG) — graf toku řízení)

Basic block je sekvence instrukcí, které se „vždy provedou za sebou“OPT. V RÁMCI JEDNÉ
FUNKCE

```

fact (int a)
{
  int D.1918;
  int ret;

  ret = 1;
  goto <D.1915>;
<D.1914>:
  ret = ret * a;
  a = a + -1;
<D.1915>:
  if (a > 1) goto <D.1914>; else goto <D.1916>;
<D.1916>:
  D.1918 = ret;
  return D.1918;
}

```

MAXIMÁLNÍ BB

BB1 LABEL

BB2

BB3

BB4

Control Flow Graph (CFG)

Definition (Control Flow Graph (CFG) — graf toku řízení)

Basic block je sekvence instrukcí, které se vždy provedou za sebou. **Control flow graph (CFG)** je orientovaný graf kde:

- 1 Vrcholy jsou basic bloky a dva speciální vrcholy: **Entry** a **Exit**.
- 2 Hrany jsou možné přechody řízení mezi basic bloky.

```
fact (int a)
{
  int D.1918;
  int ret;

  ret = 1;
  goto <D.1915>;
<D.1914>:
  ret = ret * a;
  a = a + -1;
<D.1915>:
  if (a > 1) goto <D.1914>; else goto <D.1916>;
<D.1916>:
  D.1918 = ret;
  return D.1918;
}
```

ENTRY

EXIT

Control Flow Graph (CFG)

Definition (Control Flow Graph (CFG) — graf toku řízení)

Basic block je sekvence instrukcí, které se vždy provedou za sebou. **Control flow graphs (CFG)** je orientovaný graf kde:

- 1 Vrcholy jsou basic bloky a dva speciální vrcholy: **Entry** a **Exit**.
- 2 Hrany jsou možné přechody řízení mezi basic bloky.

```
fact (int a)
{
    int D.1918;
    int ret;

    ret = 1;
    goto <D.1915>;
<D.1914>:
    ret = ret * a;
    a = a + -1;
<D.1915>:
    if (a > 1) goto <D.1914>; else goto <D.1916>;
<D.1916>:
    D.1918 = ret;
    return D.1918;
}
```

```
gcc -O2 -fdump-tree-all-graph bbs.c
dot bbs.c.012t.cfg.dot -Tpdf >cfg.pdf
```

Control Flow Graph (CFG)

Definition (Control Flow Graph (CFG) — graf toku řízení)

Basic block je sekvence instrukcí, které se vždy provedou za sebou **Control flow graphs (CFG)** je orientovaný graf kde:

- 1 Vrcholy jsou basic bloky a dva speciální vrcholy: **Entry** a **Exit**.
- 2 Hrany jsou možné přechody řízení mezi basic bloky.

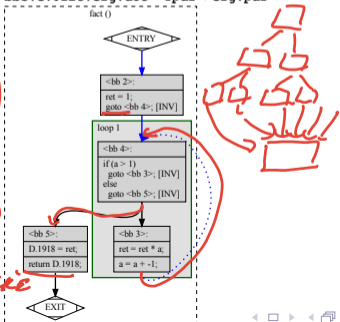
```
fact (int a)
{
  int D.1918;
  int ret;

  ret = 1;
  goto <D.1915>;
<D.1914>:
  ret = ret * a;
  a = a + -1;
<D.1915>:
  if (a > 1) goto <D.1914>; else goto <D.1916>;
<D.1916>:
  D.1918 = ret;
  return D.1918;
}
```

CFG JE DAG.
POČET HRAN JE
CCA POČET VRCHOLŮ
*2

VYSTUPNÍ STUPNĚ
≤ 2 (VĚTŠINOU)
VSTUPNÍ STUPNĚ
MŮŽOU BYT VELKÉ

```
gcc -O2 -fdump-tree-all-graph bbs.c
dot bbs.c.012t.cfg.dot -Tpdf >cfg.pdf
```



Základní skalární optimalizace

- Algebraické zjednodušování

$$A + 0 \rightsquigarrow A = A$$

FOURDUB

$$A + 0 \Rightarrow A$$

$$A + A \Rightarrow 2 \cdot A$$

$$A \cdot 3 \Rightarrow A + A + A$$

PATTERN MATCHING

$$A = (A + A) \cdot 2 \Rightarrow A = A + A$$

$$T = A + A$$

$$A = T \cdot 2$$

Základní skalární optimalizace

- Algebraické zjednodušování
- Propagace konstant

$$A = 20$$

~~$B = A + 2$~~

konst 20

$$B = 20$$

Základní skalární optimalizace

- Algebraické zjednodušení
- Propagace konstant *CONST PROP*
- Eliminace společných podvýrazů

CSE *COMMON SUBEXPRESSION
ELIMINATION*

$$A = B + C$$

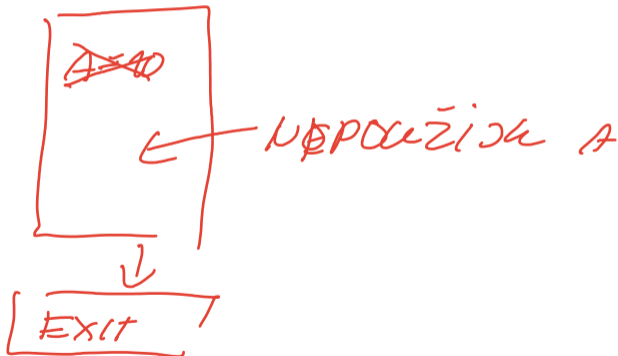
$$T = A$$

⋮

$$D = ~~B + C~~ T$$

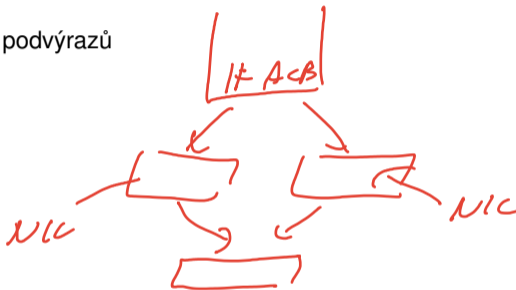
Základní skalární optimalizace

- Algebraické zjednodušování
- Propagace konstant
- Eliminace společných podvýrazů
- Mazání mrtvého kódu



Základní skalární optimalizace

- Algebraické zjednodušování
- Propagace konstant
- Eliminace společných podvýrazů
- Mazání mrtvého kódu
- Zjednodušení CFG



Základní skalární optimalizace

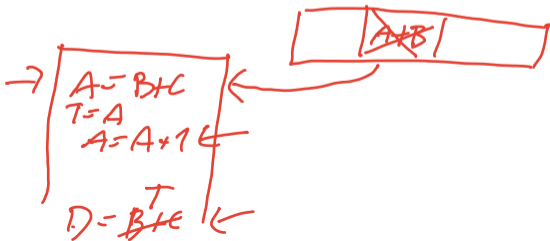
- Algebraické zjednodušování
- Propagace konstant
- Eliminace společných podvýrazů
- Mazání mrtvého kódu
- Zjednodušení CFG
- ... *COPY PROP*

Propagace konstant

Datová struktura: hash přiřazující proměným známé konstantní hodnota

Algoritmus: *SUPER*

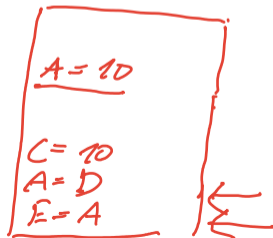
- Pro každý ~~basic block~~
 - Inicializuj prázdnou hash
 - Pro každou instrukci (statement):
 - Modifikace: Nahraď použitých proměných konstantami z hashe (pokud jsou známe) a zjednoduš výrazy
 - Invalidace: Vymaž z hashe všechny proměné, které instrukce modifikuje
 - Nastavení: Ulož do hashe všechny konstanty, které instrukce ukládá do proměných

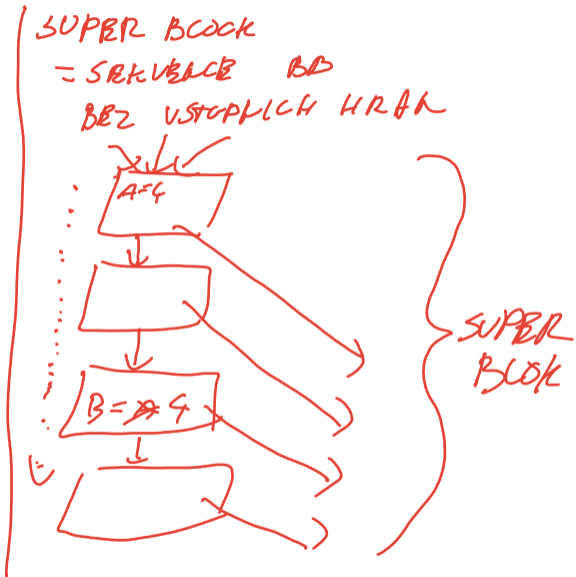
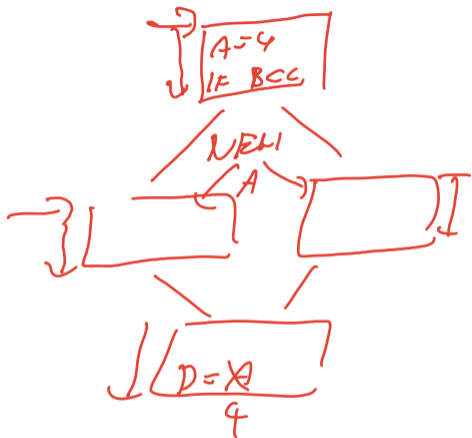


HASH



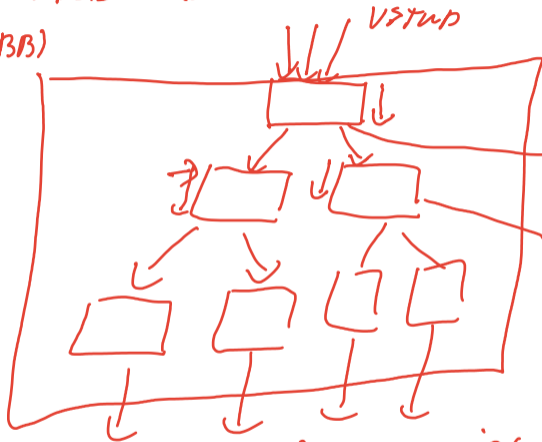
*BASIC
BLOCK*





EXTENDED BASIC BLOCK

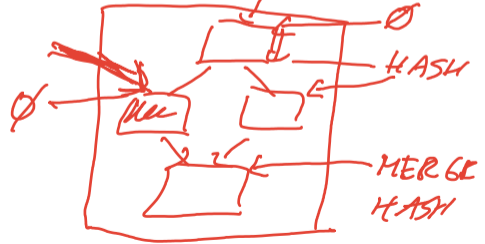
(E₂BB)



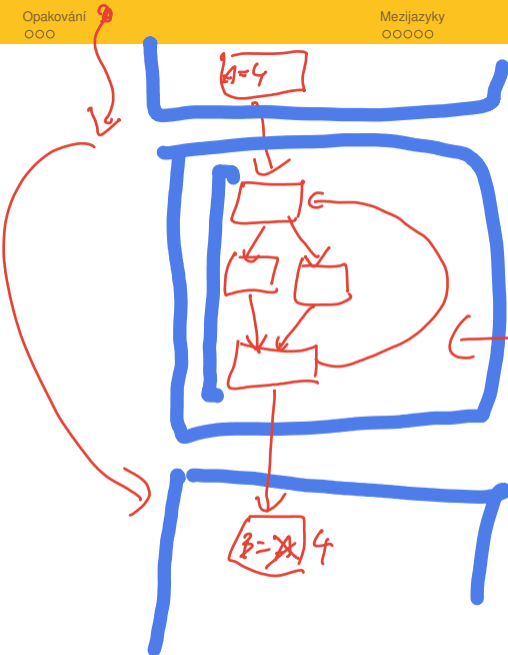
ERB

HASH

ACTUALLY REGOR



1) TOPOLOGICKÉ
POŘADÍ



✓ PŘESKAKOVÁNÍ
SMYČEK"

← A SE ZMĚNĚNÍ

PRŮSTŘE

DATAFLOW

1970's

VĚTŠINOU

LOKÁLNÍ OPT

1980's
1990's

LOKÁLNÍ OPT
+ DATAFLOW

2000
2020

SSA

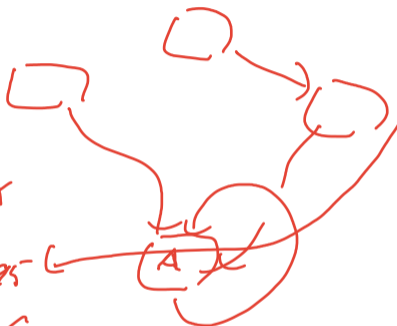
CCUR SSA

GCC 0.x

⋮
2.95

GCC 3

GCC 4+



UIC DATAFLOW

SSA

Opakování
○○○

Mezijazyky
○○○○○

CFG
○

Skalární optimalizace
○○○○○○●○○○

Opakování
○○○

Mezijazyky
○○○○○

CFG
○

Skalární optimalizace
○○○○○○○●○○

Opakování

○○○

Mezijazyky

○○○○○

CFG

○

Skalární optimalizace

○○○○○○○○●○

Opakování
○○○

Mezijazyky
○○○○○

CFG
○

Skalární optimalizace
○○○○○○○○○●