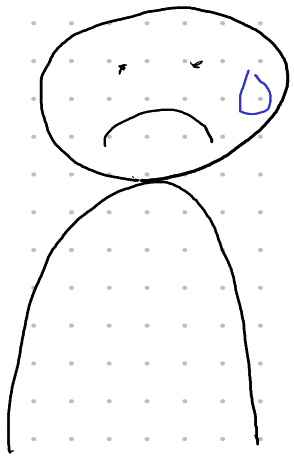
Monolithův IFJ stream
2.10.2021 v 18:00

Úvod



LL tabulka?

Precedenční tabulka?

Scanner?

Syntaktická analýza?

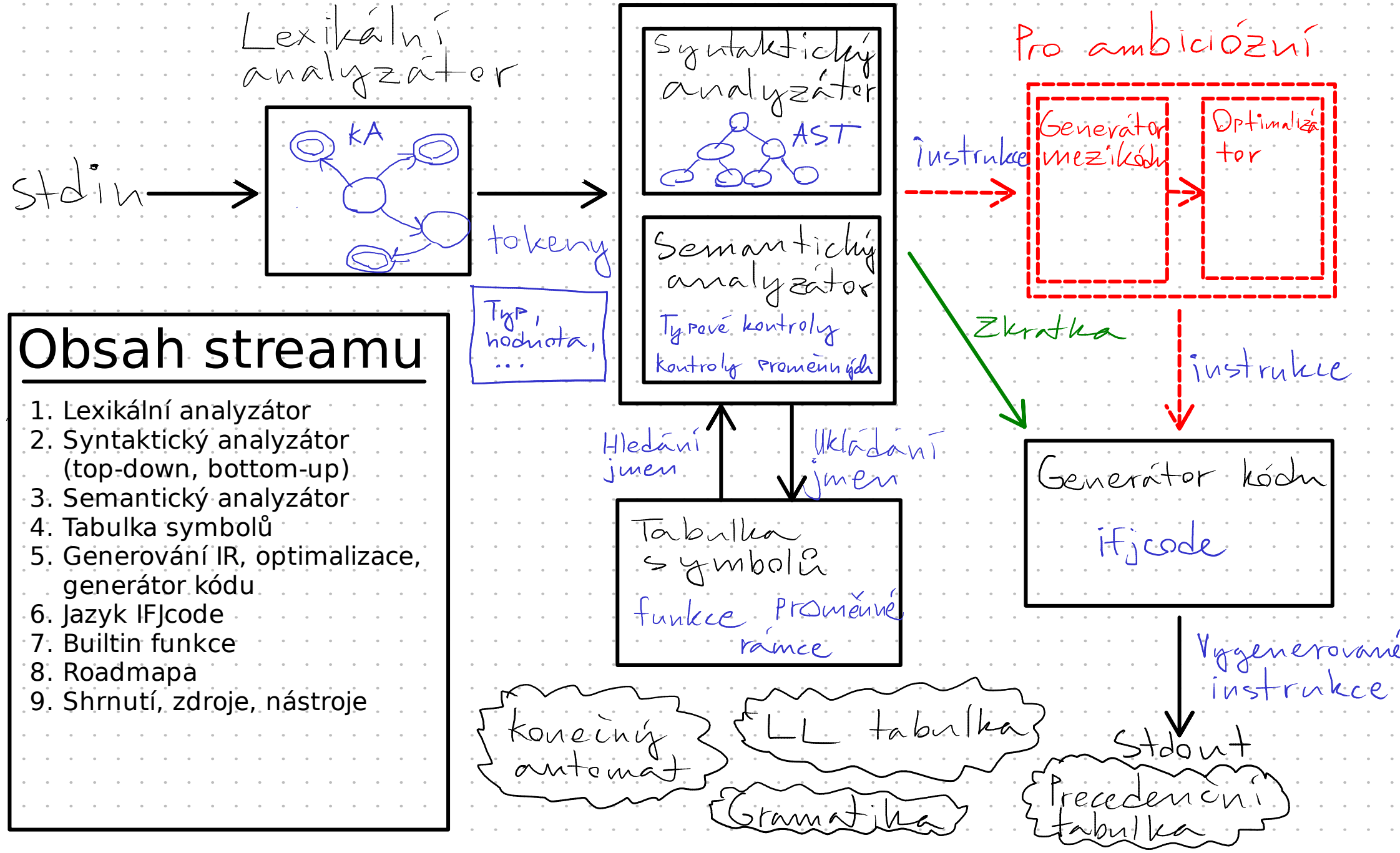
Tabulka symbolů?

Top-down?

Sémantické kontroly?!?

Bottom-up?

High level view

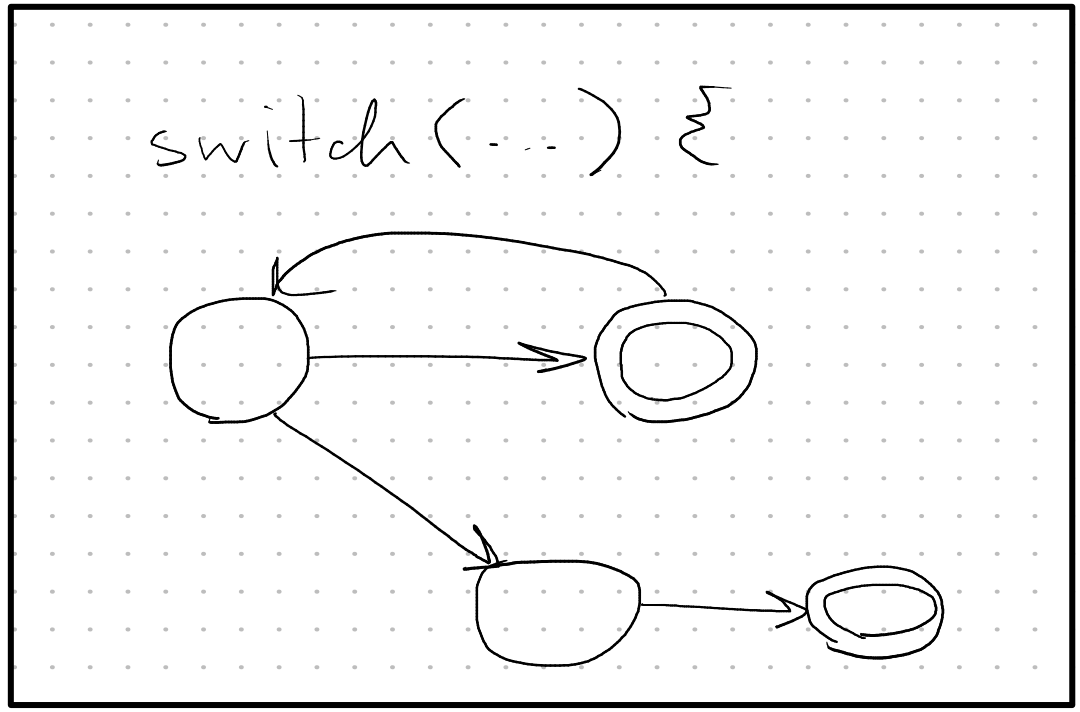


Obsah streamu

1. Lexikální analyzátor
2. Syntaktický analyzátor (top-down, bottom-up)
3. Semantický analyzátor
4. Tabulka symbolů
5. Generování IR, optimalizace, generátor kódu
6. Jazyk IFJcode
7. Builtin funkce
8. Roadmapa
9. Shrnutí, zdroje, nástroje

Lexikální analyzátor (the easy part)

Nejdříve definovat KA, potom implementovat



Vstup →

→ tokeny ...

Pozor na neočekávaný EOF a jiné edge cases

Token
- typ (konstanta, keyword, identifikátor, ...)
- hodnota
- Ladící informace (začátek, konec, číslo řádky ...)

Syntaktický analyzátor (the hard part)

Top-down

Rekurzivní

⊕ Menší režie při implementaci

⊖ Při změnách v gramatice se obtížněji upravuje

×

Prediktivní

⊕ Snadné úpravy gramatiky, nemá rekurzi

⊖ Obtížnější implementace

Bottom-up

- Precedenční syntaktická analýza („shift-reduce parser")
- Pro zpracování výrazů

AST?

- Někdy se bez něj lze obejít (např. pouze pro výrazy)
- Doporučuji generovat AST pro celý program (ušetří hodně starostí)
- IAL zanáška – inorder, preorder a postorder se při průchodu AST velmi hodí

Rekurzivní sestup

Gramatika

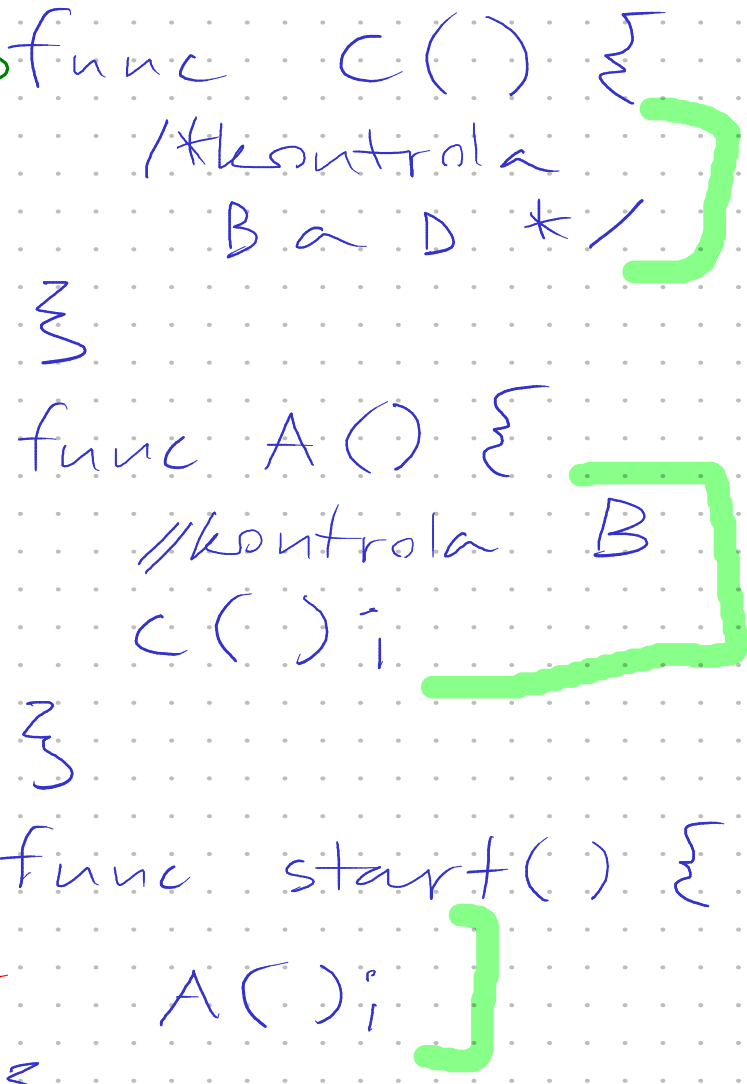
START: A;

A: B C;

C: D B;

- Pravidla gramatiky se „mapují" na funkce

Kód



⚠ Čím kvalitnější gramatiku vytvoříte, tím méně předělávek později

Precedenční analýza

– Řeší to, s čím si top-down parser neporadí

$a * 1 + c * 3$

zásobník	vstup	strom
$	a * 1 + c * 3 <	
$ a	* 1 + c * 3 > 3.	
$ E	* 1 + c * 3 <	E
$ < E *	1 + c * 3 <	E
$ < E * 1	+ c * 3 > 3.	E * 1
$ < E * E	+ c * 3 > 2.	E * E
$ E		
	....	
$	$!!	

vstup

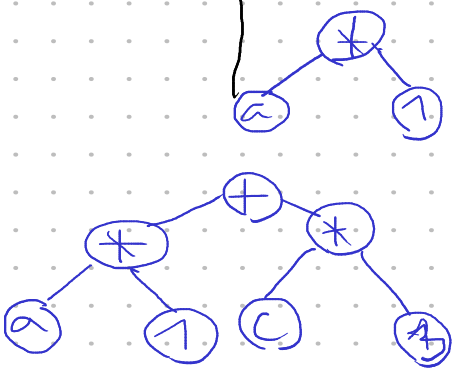
	+	*	i	$
+	>	<	<	>
*	>	>	<	>
i	>	>		>
$	<		<	!!

zásobník

1: E → E + E

2: E → E * E

3: E → i



Ping-pong

top-down
bottom-up

```
func x() {
    if 1 + 1 == 2 {
        //code...
    }
}
```

- Top-down a bottom up parsery si průběžně předávají kontrolu

Semantický analyzátor

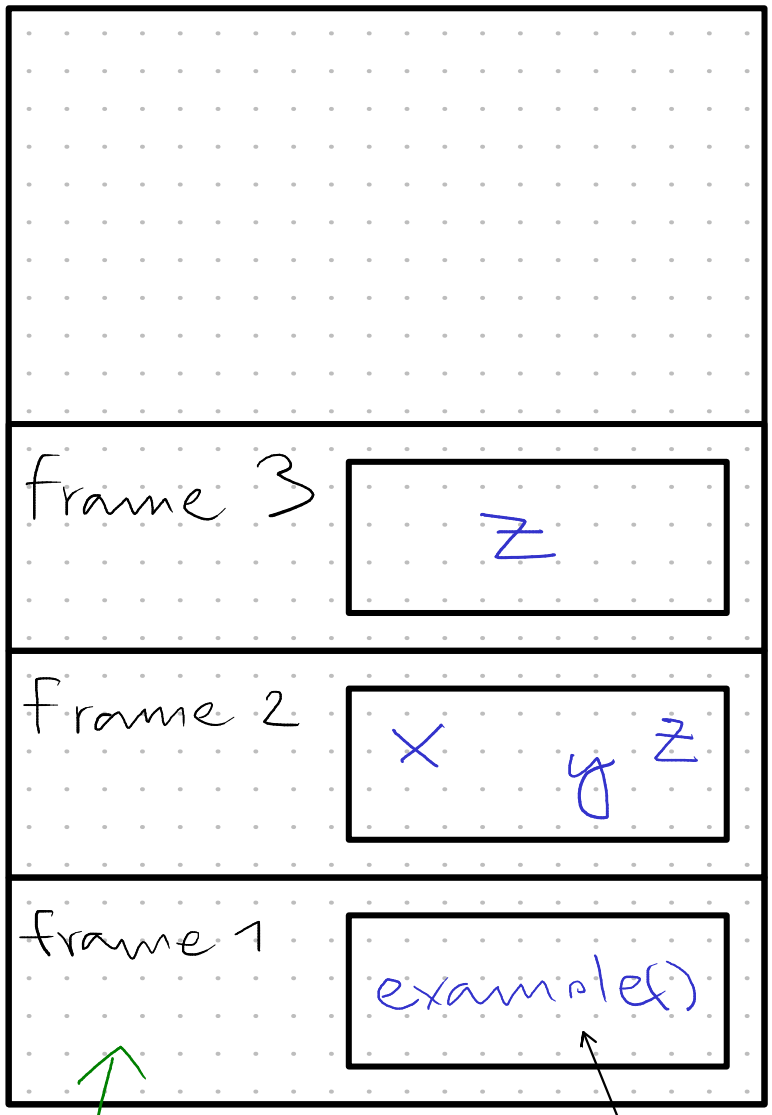
- Ne nutně jasně vyhraněná komponenta!
  (sémantické kontroly „rozdrobeny" v syntaktických analyzátorech)

Vrací funkce co má? Vrací v každé větvi?

Sedí typy? Jsou proměnné definovány?

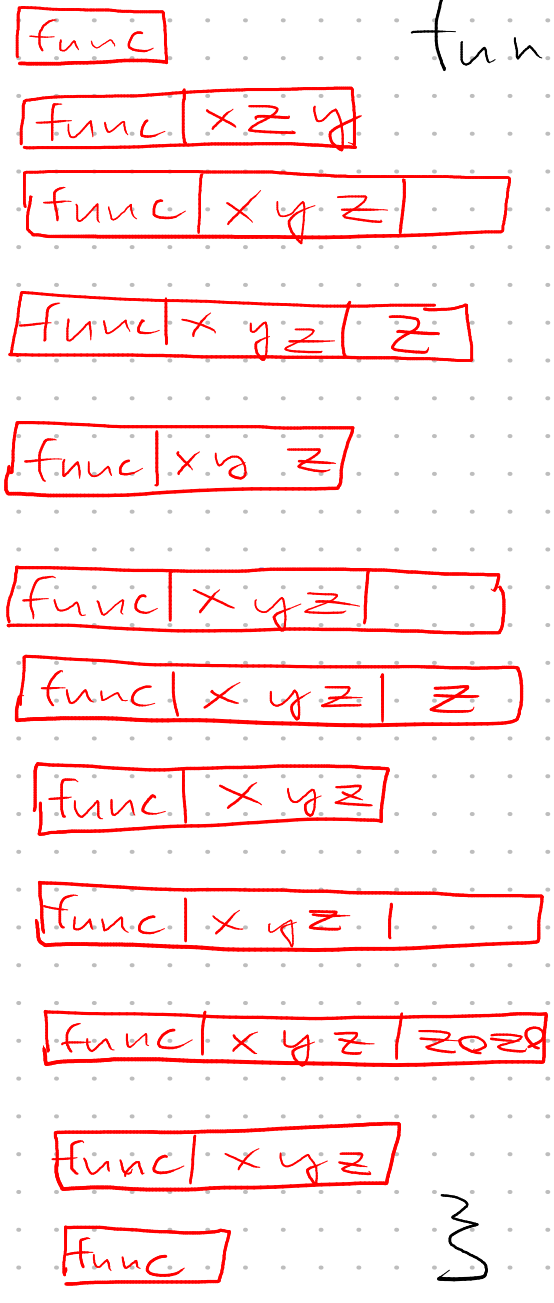
Nejsou redefinovány?

Tabulka symbolů



Globální rámec

BVS nebo hash table




```
func example(x, y) {
    z := b
    if x == 42 {
        z := 1337
    }
    if y == 5 {
        z := 420
    }
    while 1 {
        zozo := 3
    }
}
```

Pozor na aliasing

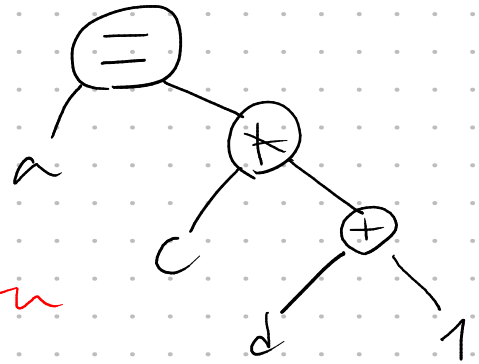
Pozor na defvar v cyklu

Pozor na name mangling

Generování IR, optimalizace, generátor kódu

$$a = c * (d + 1)$$

Registrový (3AK...) ×



Zásobníkový

⊕ Méně instrukcí

⊖ Potřeba více kontextu při generování

⊕ Jednodušší impl., méně adres

⊖ Více instrukcí, musíme řešit pořadí generování

```
ADD LF@a LF@d int@1 //d+1
MUL LF@a LF@a LF@c //c*(d+1)
```

```
PUSHS LF@c
PUSHS LF@d
PUSHS int@1
ADDS  //d+1
MULS  //c*d+1
POPS  LF@a //a=vyst
```

▽ Pozor na pořadí

- 3AK vhodnější na optimalizaci
- MEZIKÓD A OPTIMALIZÁTOR JSOU VOLITELNÉ? (lze i přímo „printovat")
  - konkrétní techniky na přednáškách

Návraty hodnot, variadické funkce

– Zde se hodí intuice z ISU?

```
func f(x, y, z) {
    return z, y, x
}

a, b, c := f(1, 2, 3)
```

```
(PUSH  int@3)      argc pro variadické funkce
PUSH   int@3       něco jako
PUSH   int@2       decl...
PUSH   int@1
JUMP   f
//vrátí pomocí
PUSHS  LF@x
PUSHS  LF@y
PUSHS  LF@z
POPS   LF@a
POPS   LF@b
POPS   LF@c
```

Jazyk IFJcode

Práce s rámci, volání funkcí

- Při vytvoření rámce nemůžete přistoupit ke starším rámcům

Práce s datovým zásobníkem

- Předávání hodnot mezi funkcemi, jinak zejména při zásobníkovém kódu

Aritmetické, relační, bool, ...

- Na výběr mezi registrovými a zásobníkovými variantami

Práce s řetězci

- Nemají zásobníkovou variantu!

Řízení toku programu

- konstrukce podobné jak v ISU

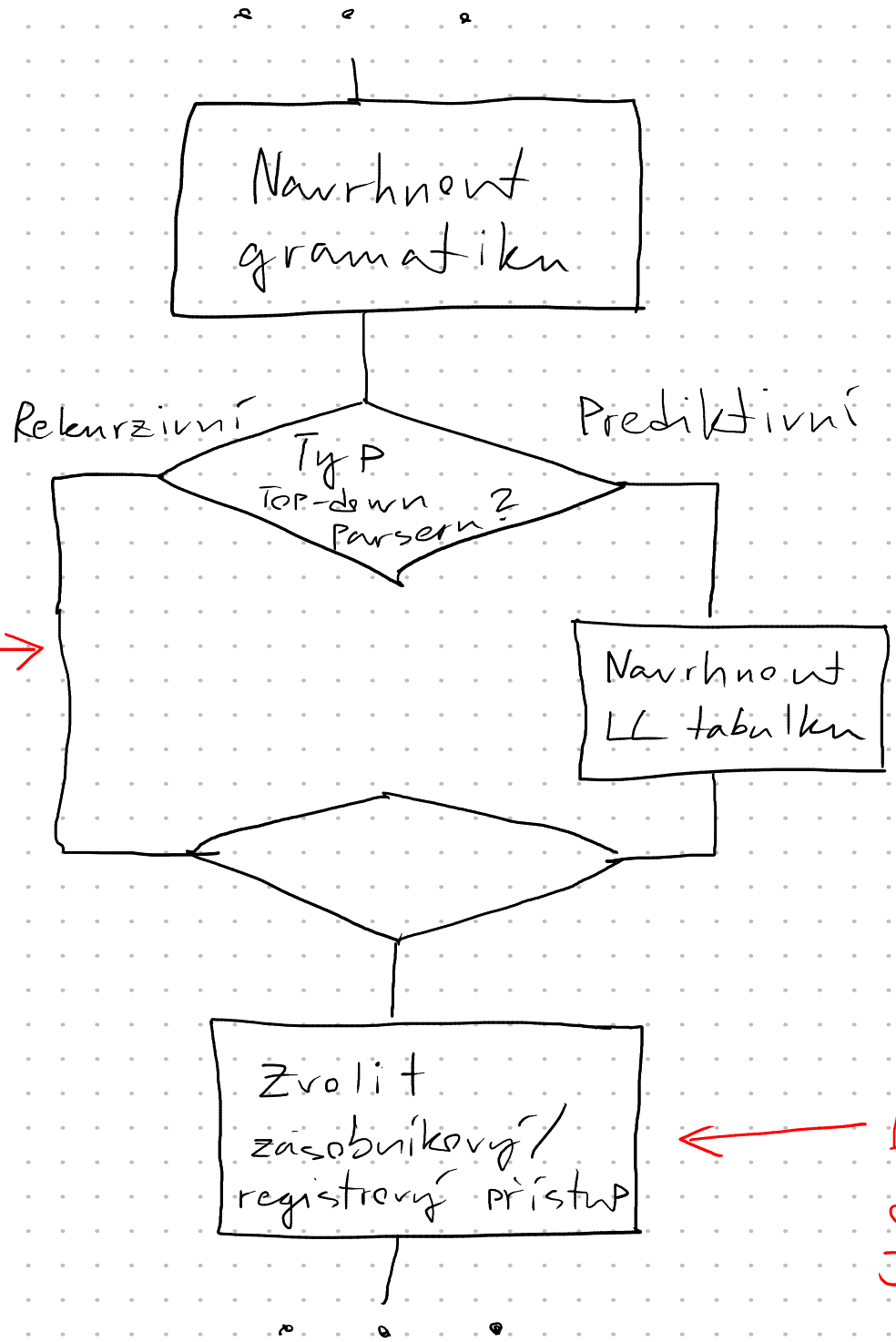
⚠ Nemusíte použít zdaleka všechny funkce IFJcode (implementujte pouze to, co potřebujete)

Builtin funkce

- Napsat přímo v IFJcode, injektovat do generovaného kódu
- V tabulce symbolů by měly být názvy rezervované
- Mohou být potřeba variadické funkce

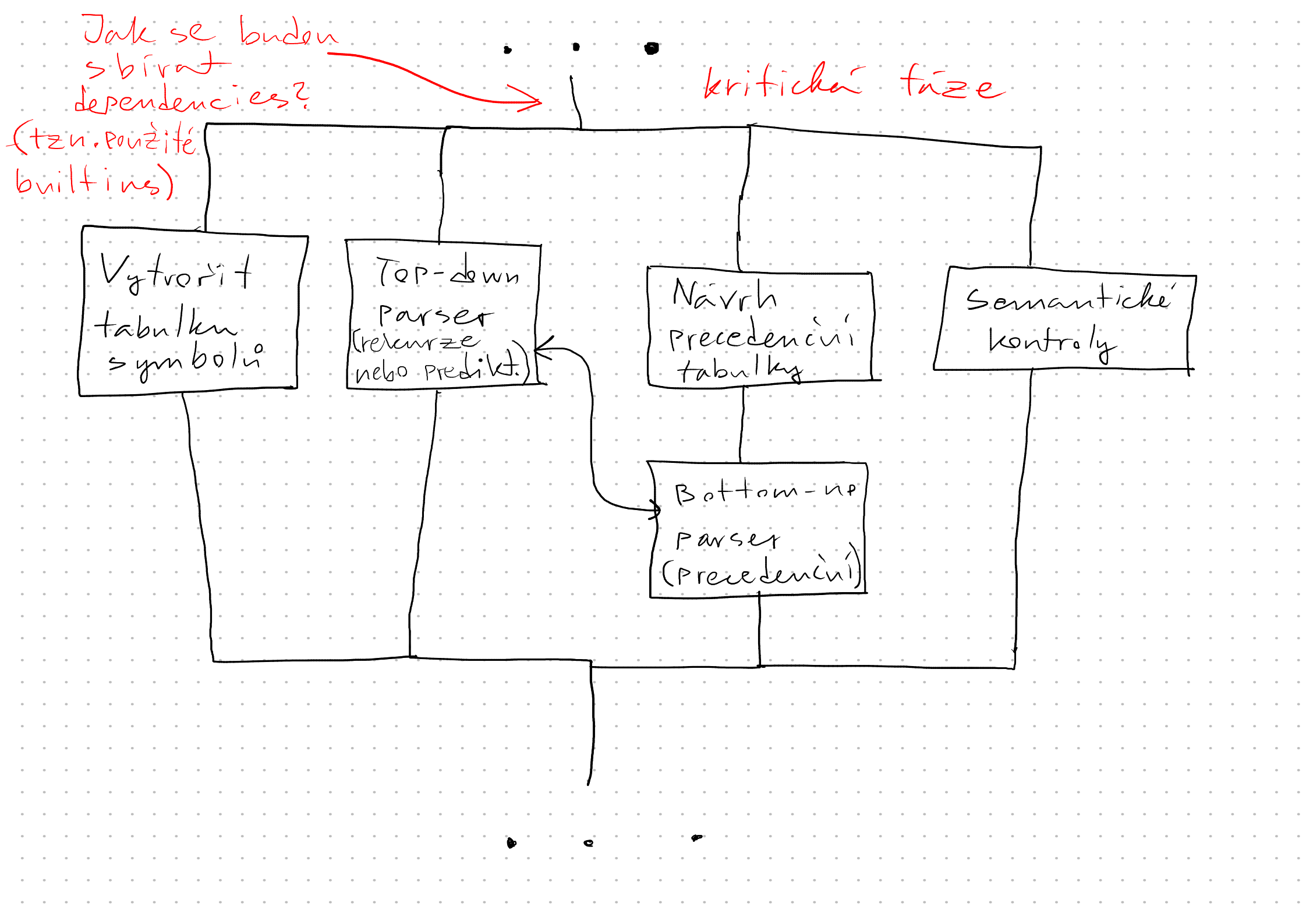
← může dosti ovlivnit výslednou implementaci.

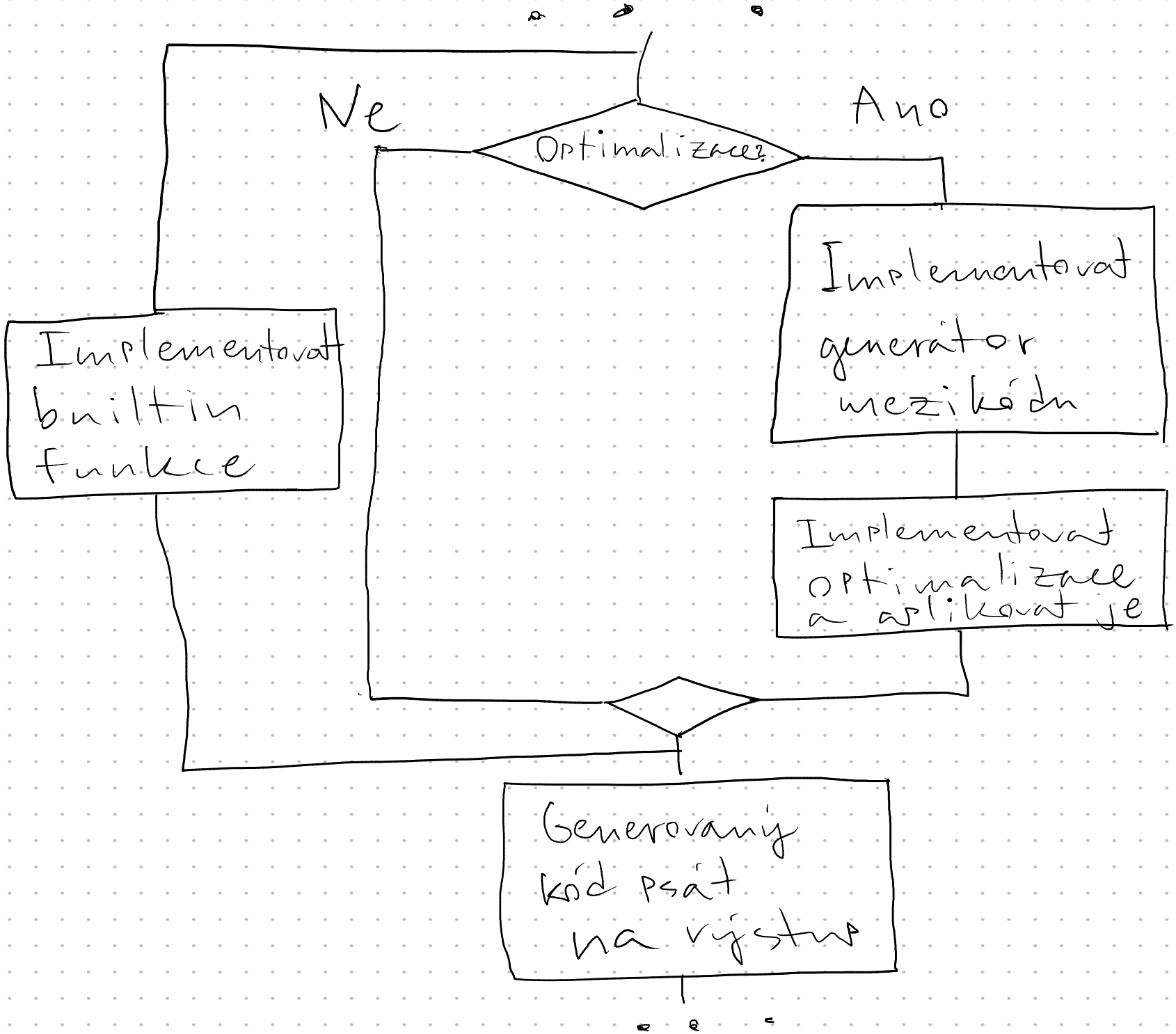
scanner = lex. analyzátor

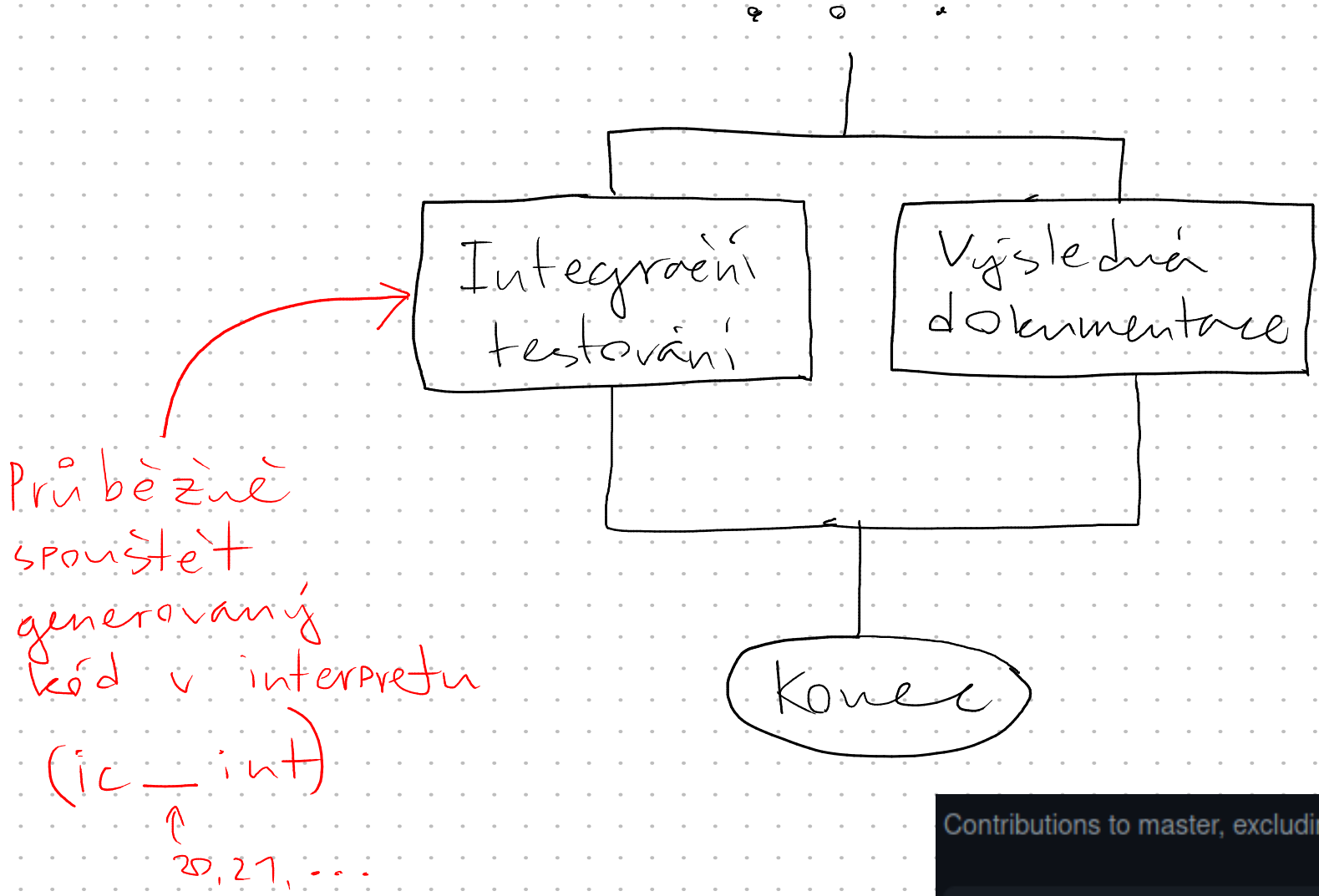


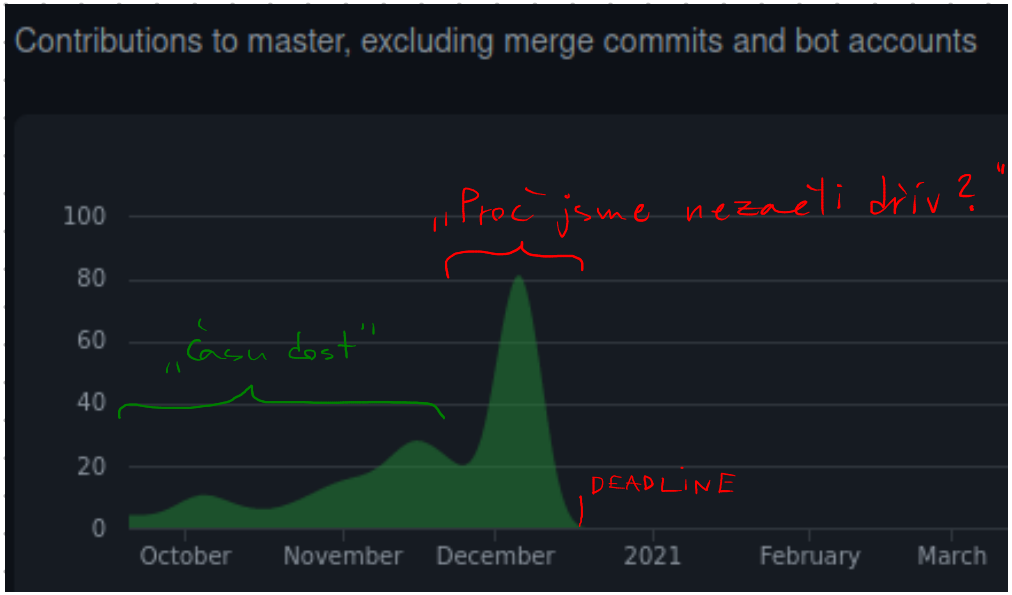
LL tabulku lze navrhnout kdykoliv...

Bude se dělat optimalizace? Jak se co bude generovat?


Jak se budou sbírat dependencies? (tzn. použité builtiny)
kritická fáze
Vytvořit tabulku symbolů
Top-down parser (rekurze nebo predikt.)
Návrh precedenční tabulky
Sémantické kontroly
Bottom-up parser (precedenční)


Optimalizace?
Ne
Ano
Implementovat builtin funkce
Implementovat generátor mezikódu
Implementovat optimalizace a aplikovat je
Generovaný kód psát na výstup


Integrační testování
Výsledná dokumentace
Konec
Průběžně spouštět generovaný kód v interpretu (ic — int)
20, 21, ...


Contributions to master, excluding merge commits and bot accounts
100
80
60
40
20
0
October
November
December
2021
February
March
„Proč jsme nezačali dřív?"
„Času dost"
DEADLINE

Shrnutí, zdroje, nástroje

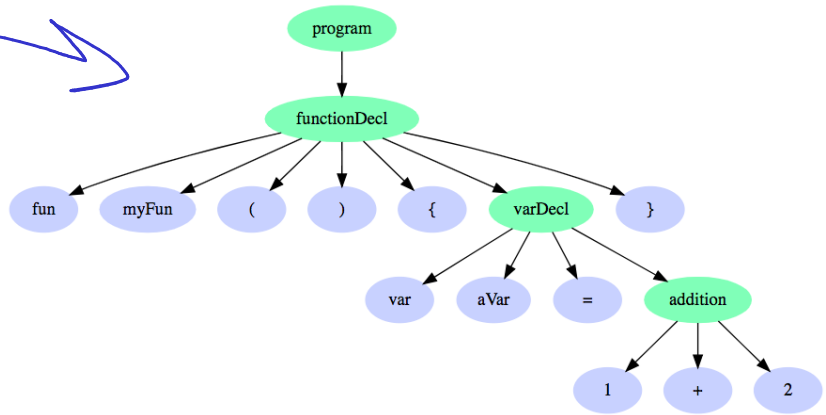
Debug (printy mi často stačit nebudou...)

- valgrind, callgrind, ...
- gdb ve VS Code
  - -g -O0
  - -fsanitize...

Leaky „nevadí" dokud interpret pracuje správně

Někdy je lepší tužka než debugger

- GraphViz pro debug/vizualizaci parsingu (např. zobrazení AST)



Zdroj: https://gist.github.com/pepasflo/04f3c440b824a638fe9559494597f3b0

Studijní materiály

- Prezentace IFJ...
- https://craftinginterpreters.com/
  vizuálně přívětivé
- Dragon book (těžší kalibr)
  https://suif.stanford.edu/dragonbook/

- Vyhněte se githubu (plagiát a neoptimální řešení...)

Testování

- Unit testy např. Google test
- Integrační testy: přeložit a porovnat výstupy s referenčním...

Rady do života

- Bez znalosti teorie se velmi špatně spolupracuje
- Přemýšlet hodně před implementací než zpětně...
- Čím více test cases, tím lépe
- Všechno na streamu jsou pouze subjektivní názory, je to VÁŠ projekt a vlastní nápaditou implementaci si můžete opravdu užít!