Operační systémy a databáze

Petr Štěpán, K13133

KN-E-229

stepan@labe.felk.cvut.cz

Téma 14. Od algoritmu k procesu

Příprava a zpracování programů

- Při psaní programu je prvním krokem (pomineme-li analýzu zadání, volbu algoritmu atd.) vytvoření zdrojového textu ve zvoleném programovacím jazyku
  - zpravidla vytváříme textovým editorem a ukládáme do souboru s příponou indikující programovací jazyk
    - `zdroj.c` pro jazyk C
    - `prog.java` pro jazyk Java
    - `text.cc` pro C++
- Každý takový soubor obsahuje úsek programu označovaný dále jako modul
  - V závislosti na typu dalšího zpracování pak tyto moduly podléhají různým sekvencím akcí, na jejichž konci je jeden nebo několik výpočetních procesů
- Rozlišujeme dva základní typy zpracování:
  - Interpretace
  - Kompilace (překlad)

  - existuje i řada smíšených přístupů

Interpretace programů

- Interpretem rozumíme program, který provádí příkazy napsané v nějakém programovacím jazyku
  - vykonává přímo zdrojový kód
    - mnohé skriptovací jazyky a nástroje (např. bash), historické verze BASIC
  - překládá zdrojový kód do efektivnější vnitřní reprezentace a tu pak okamžitě „vykonává“
    - jazyky typu Perl, Python, MATLAB apod.
  - vykonává předkompilovaný a uložený kód, vytvořený překladačem, který je součástí interpretačního systému
    - Java (překládá se do tzv. „byte kódu“ interpretovaného JVM)
- Výhody:
  - rychlý vývoj bez potřeby explicitního překladu a dalších akcích
  - praktická nezávislost na cílovém stroji
- Nevýhody:
  - nízká efektivita „běhu programu“, neboť interpret stále analyzuje zdrojový text (např. v cyklu)
- Důležitý poznatek:
  - binární strojový kód je vždy interpretován hardwarem

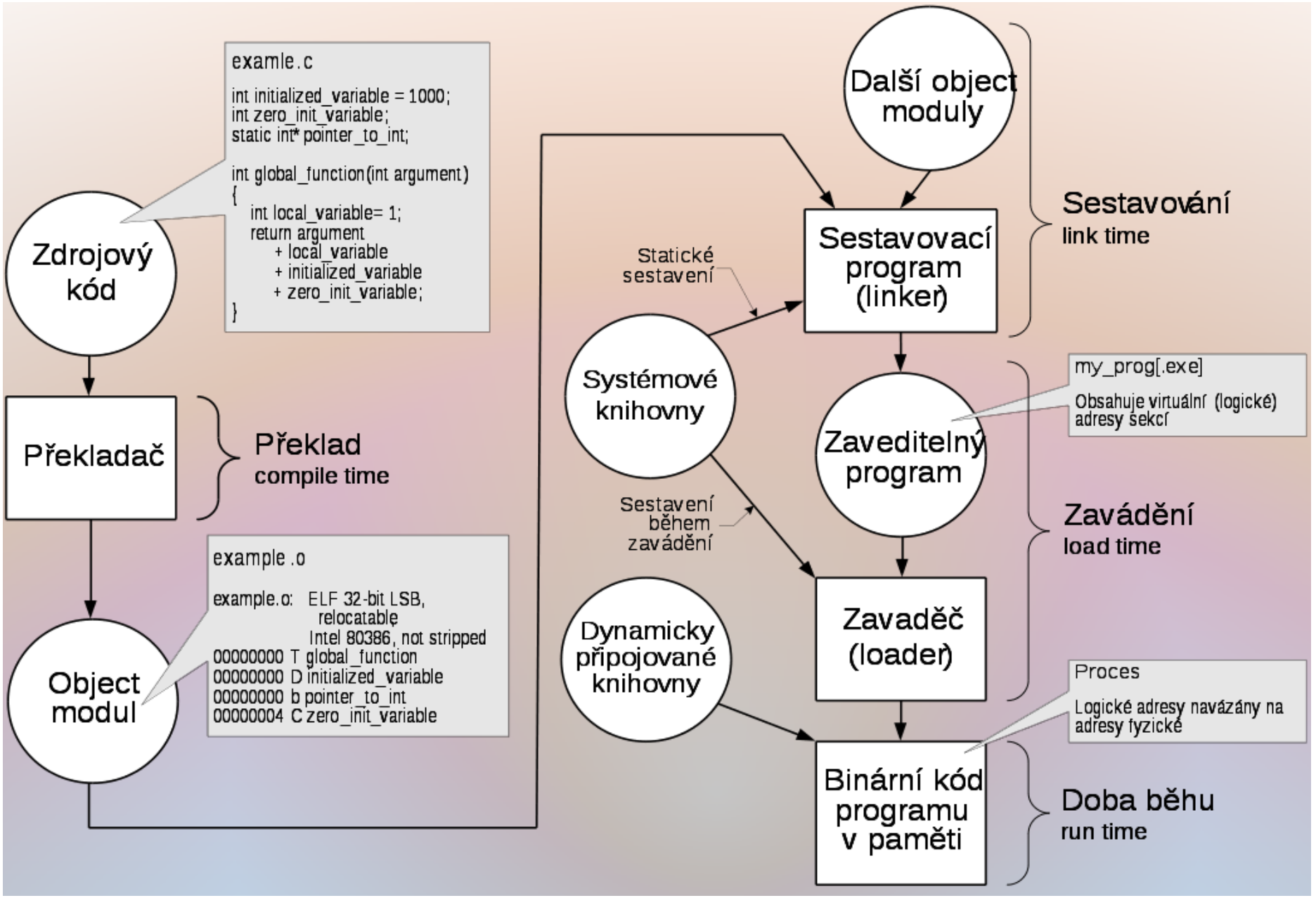
Překladač (*Compiler*)

- Úkoly překladače (kompilátoru)
  - kontrolovat správnost zdrojového kódu
  - „porozumět“ zdrojovému textu programu a převést ho do vhodného „meziproduktu“, který lze dále zpracovávat bez jednoznačné souvislosti se zdrojovým jazykem
  - základní výstup kompilátoru bude záviset na jeho typu
    - tzv. **nativní překladač** generuje kód stroje, na kterém sám pracuje
    - **křížový překladač** (*cross-compiler*) vytváří kód pro jinou hardwarovou platformu (např. na PC vyvíjíme program pro vestavěný mikropočítač s procesorem úplně jiné architektury, než má naše PC)
  - mnohdy umí překladač generovat i ekvivalentní program v jazyku symbolických adres (*assembler*)
  - častou funkcí překladače je i optimalizace kódu
    - např. dvě po sobě jdoucí čtení téže paměťové lokace jsou zbytečná
    - jde často o velmi pokročilé techniky závislé na cílové architektuře, na zdrojovém jazyku
    - optimalizace je časově náročná, a proto lze úrovně optimalizace volit jako parametr překladu
      - při vývoji algoritmu chceme rychlý překlad, při konečném překladu provozní verze programu žádáme rychlost a efektivitu

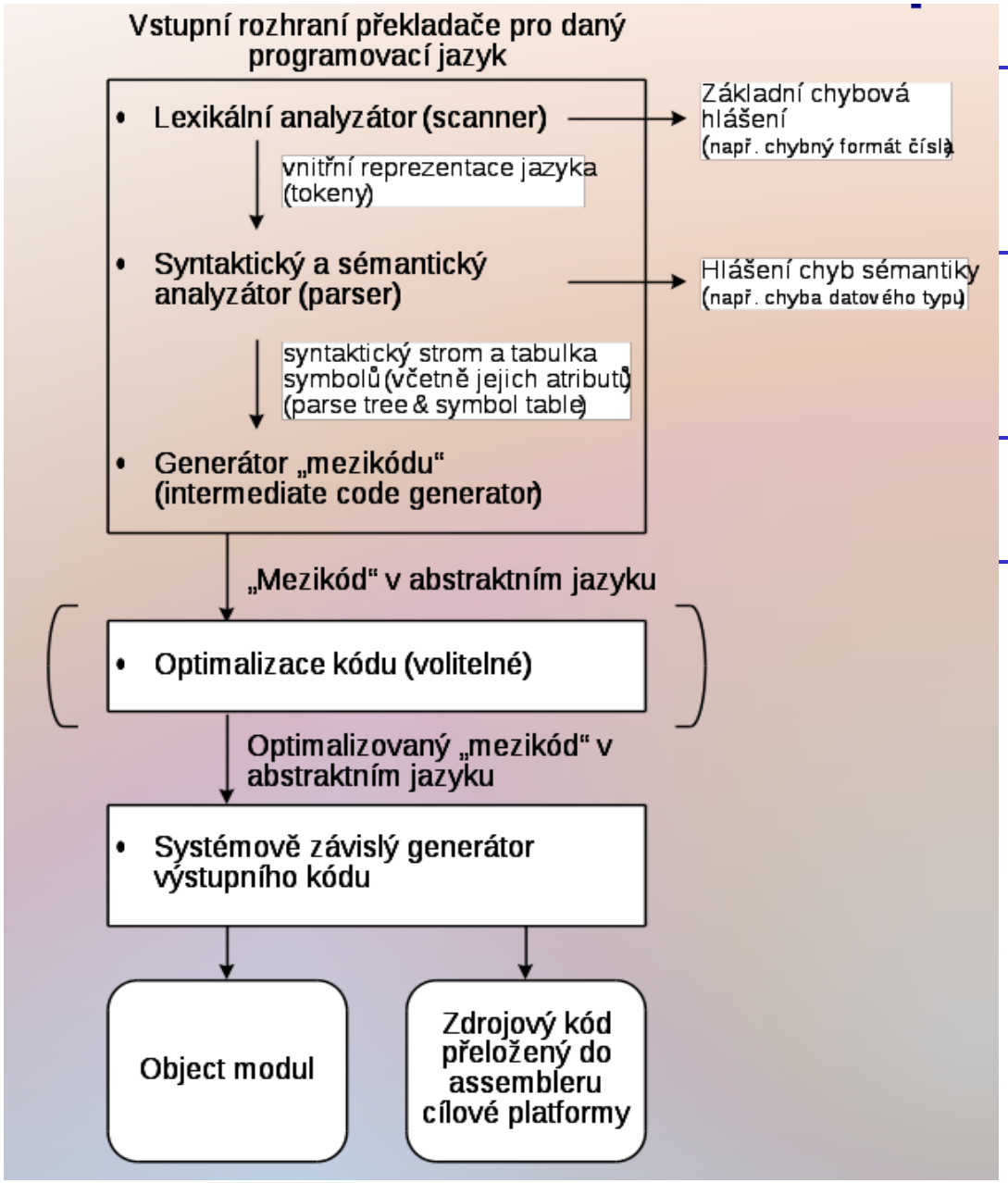
Rámcová činnost překladače

- Překlad programu probíhá v řadě fází (kroků)
- Lexikální analýza
  - převádí textové řetězce na série *tokenů* (též lexemů), tedy textových elementů detekovaného typu
  - např. příkaz: `sedm = 3 + 4` generuje tokeny
    [sedm, IDENT], [=, ASSIGN_OP], [3, NUM], [+, ADD_OP], [4, NUM]
  - Již na této úrovni lze detekovat chyby typu „nelegální identifikátor“
    (např. `1q`)
  - Tvorbu lexikálních analyzátorů lze mechanizovat pomocí programů typu lex nebo flex
- Syntaktická analýza
  - kontroluje, zda sekvence tokenů odpovídají legálním pravidlům, která popisují příslušný programovací jazyk (zpravidla vyjádřeno bezkontextovou gramatikou)
  - zde lze detekovat chyby typu „nedefinovaný identifikátor“; „číslu se přiřazuje řetězec“ apod.
- Sémantická analýza
  - vytváří příslušné posloupnosti akcí definujících, jak chápat (a posléze vykonat) syntaktickou analýzou nalezené abstraktní výrazy
  - výsledkem analýzy jsou zpravidla hierarchické stromové struktury (*parse tree*) [parse = „větný rozbor“]

Od zdrojového textu k procesu



Struktura překladače



- Detailní struktura překladače závisí na konkrétním zdrojovém jazyku
- „Mezikód“ většinou reprezentuje tzv. syntaktický strom překládaného programu
- Optimalizaci lze většinou vypnout
- Generovaný kód závisí na dalším způsobu zpracování systémem
  - některé překladače generují výhradně assembler a ten je pak převáděn do cílové binární formy

Rámcová činnost překladače (pokr.)

- Syntaktická a sémantická analýza
  - většinou bývá prováděna společným kódem překladače, zvaným parser
  - Tvorba parserů se mechanizuje pomocí programů typu yacc či bison
    - yacc = Yet Another Compiler Compiler; bison je zvíře vypadající jako yacc
- Programovací jazyky se formálně popisují vhodnou deskripční metodou
  - např. pomocí tzv. Backus-Naurovy Formy (BNF)
  - BNF – elementární příklad české soukromé poštovní adresy:

```
<postal-address> ::= <name-part> <street-address> <psc-part>
<name-part>        ::= <personal-part> <last-name> <EOL>
                     | <personal-part> <name-part>
<personal-part>  ::= <first-name>
                     | <initial> "."
<street-address> ::= <street-name> <house-num> <EOL>
<psc-part>         ::= <PSC> “  “ <town-name> <EOL>
```

rekurze

Rámcová činnost překladače (pokr.)

- Častěji se užívá EBNF (Extended BNF)
  - obvyklá symbolika EBNF
  - Mnohdy různé dialekty EBNF

Užití	Notace	Užití	Notace
definice	=	opakování	{ ... }
zřetězení	,	grupování	(...)
zakončení	.	terminální řetězec	“ ... “
alternativa	|	terminální řetězec	‘ ... ‘
volitelné	[...]	komentář	(* ... *)

  - Příklad užití EBNF

```
digit_excluding_zero = "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
digit                = "0" | digit_excluding_zero.
twelve               = "1","2".
three_hundred_twelve = "3",twelve.
natural_number       = digit_excluding_zero,{digit}.
integer              = "0" | ["-"], natural_number.
arit_operator        = "+" | "-" | "*" | "/".
simple_int_expr      = integer,arit_operator,integer.
```

  - EBNF pro jazyk C lze nalézt na http://www.cs.man.ac.uk/~pjj/bnf/c_syntax.bnf
  - EBNF pro jazyk Java http://cui.unige.ch/isi/bnf/JAVA/AJAVA.html

Optimalizace během překladu

- Co může překladač optimalizovat
- Elementární optimalizace
  - předpočítání konstant
    - `n = 1024 * 64` – během překladu se vytvoří konstanta 65536
  - znovupoužití vypočtených hodnot
    - `if(x**2 + y**2 <= 1) a = x**2 + y**2 else a=0;`
  - detekce a vyloučení nepoužitého kódu
    - `if((a>=0) && (a<0)) { never used code; };`
    - obvykle se generuje „upozornění“ (warning)
- Sémantické optimalizace
  - značně komplikované
  - optimalizace cyklů
    - lepší využití principu lokality (viz téma o správě paměti)
    - minimalizace skoků v programu – lepší využití instrukční cache
  - ...
- Celkově mohou být optimalizace
  - velmi náročné během překladu, avšak za běhu programu mimořádně účinné (např. automatická paralelizace)

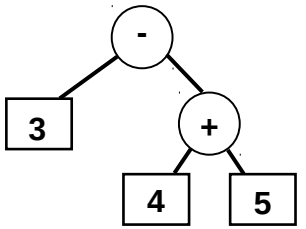
Generátor kódu

- Generátor kódu vytváří vlastní sémantiku "mezikódu"
  - Obecně: Syntaktický a sémantický analyzátor buduje strukturu programu ze zdrojového kódu, zatímco generátor kódu využívá tuto strukturální informaci (např. datové typy) k tvorbě výstupního kódu.
  - Generátor kódu mnohdy dále optimalizuje, zejména při znalosti cílové platformy
    - např.: Má-li cílový procesor více střádačů (datových registrů), dále nepoužívané mezivýsledky se uchovávají v nich a neukládají se do paměti.
- Podle typu překladu generuje různé výstupy
  - assembler (jazyk symbolických adres)
  - absolutní strojový kód
    - pro „jednoduché“ systémy (firmware vestavných systémů)
  - přemístitelný (object) modul ➔
  - speciální kód pro pozdější interpretaci virtuálním strojem
    - např. Java byte-kód pro JVM
- V interpretačních systémech je generátor kódu nahrazen vlastním „interpretem“
  - ukážeme několik principů používaných interprety (a někdy i generátory cílového kódu)

Výrazy v postfixovém zápisu

- Postfixová notace (též reverzní polská notace, RPN)
  - 1920 představeno polským matematikem Łukasiewiczem
  - operátor následuje své operandy, přičemž je odstraněna nutnost používat závorky (priorita operátorů se vyjadřuje samotným zápisem výrazu)
  - např. „infixové“ `3-4+5=(3-4)+5` v je RPN `3_4_-_5_+`
  - zatímco `3-(4+5)` bude v RPN `3_4_5_+_-`
- Výpočet RPN výrazů zásobníkovým automatem

Stromová reprezentace výrazu 3-(4+5)



- Dokud jsou na vstupu znaky
  - přečti další znak
  - jestliže je znak hodnota
    - ulož ji na zásobník
  - jinak znak představuje funkci (operátor)
    - je známo, že funkce přebírá ***n*** parametrů
    - vyber ze zásobníku ***n*** hodnot
      - jestliže je na zásobníku méně než ***n*** hodnot
        - ***(Chyba)*** dostatečný počet parametrů
    - vypočti hodnotu funkce
    - výsledek ulož zpět na zásobník
- jestliže je na zásobníku jen jedna hodnota
  - je to výsledek výpočtu
- jinak
  - ***(Chyba)*** příliš mnoho hodnot

Výrazy v postfixovém zápisu (pokr.)

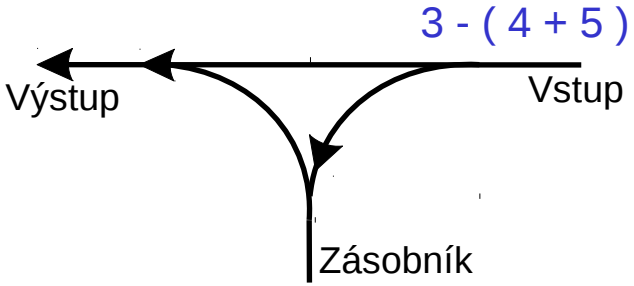
- Výpočet - příklad

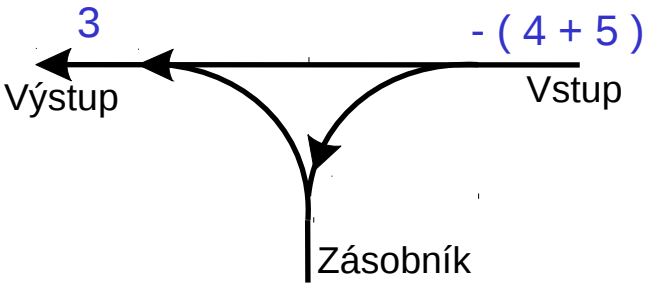
`8+((5+2)*7)-3` je v RPN `8_5_2_+_7_*_+_3_-`

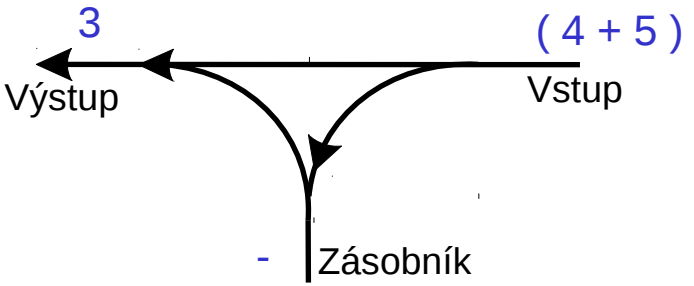
Vstup	Akce	Zásobník
8	Ulož na zásobník	8
5	Ulož na zásobník	8, 5
2	Ulož na zásobník	8, 5, 2
+	Dvě hodnoty z vrcholu zásobníku sečti a výsledek ulož zpět	8, 7
7	Ulož na zásobník	8, 7, 7
*	Dvě hodnoty z vrcholu zásobníku násob a výsledek ulož zpět	8, 49
+	Dvě hodnoty z vrcholu zásobníku sečti a výsledek ulož zpět	57
3	Ulož na zásobník	57, 3
-	Dvě hodnoty z vrcholu zásobníku odečti a výsledek ulož zpět	54

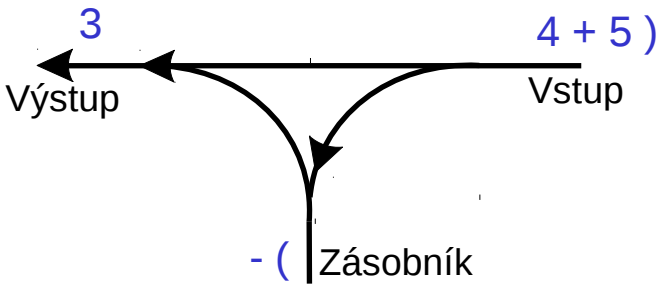
Přepis infixového zápisu do RPN

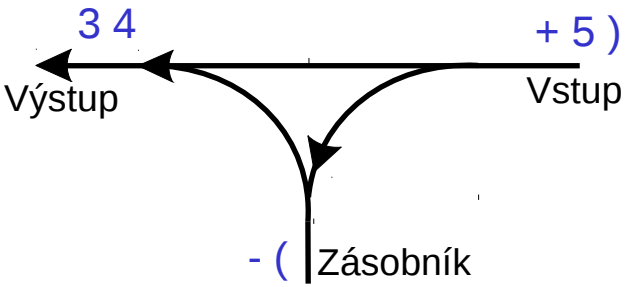
- Dijkstrův „*Shunting yard*“ algoritmus
  - doslovný překlad „algoritmus seřaďovacího nádraží“

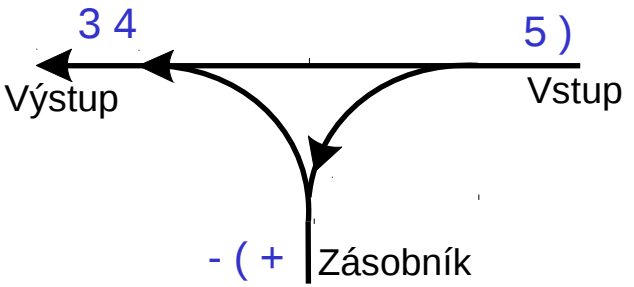


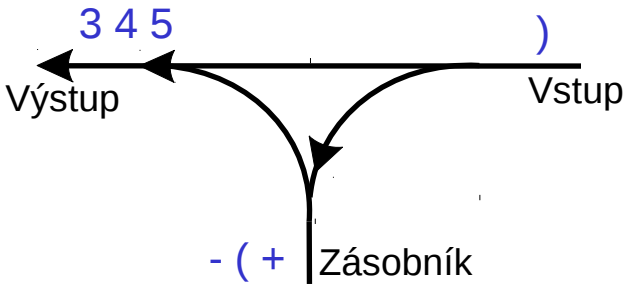


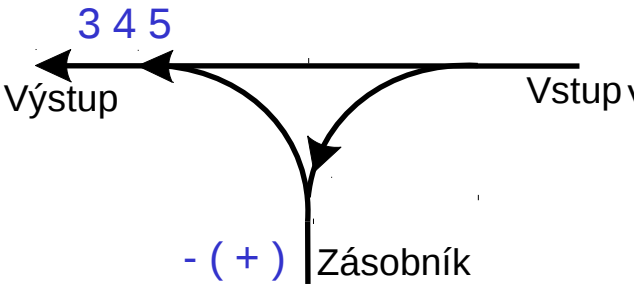


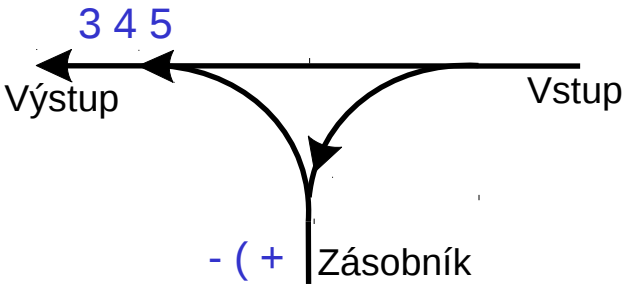


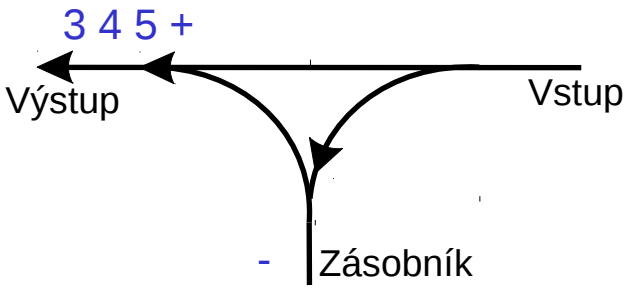


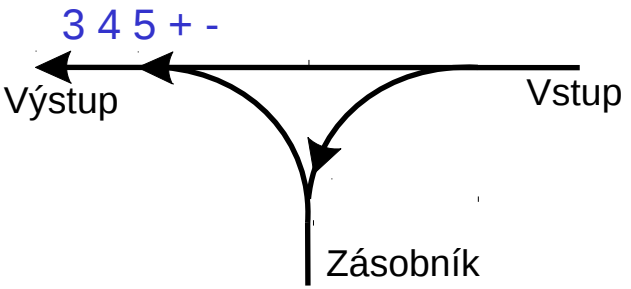












Algoritmus „Shunting yard“ detailně

- Dokud jsou na vstupu tokeny:
  - Přečti token.
  - Pokud je token číslo či proměnná, předej ho na výstup.
  - Pokud je token funkce, ulož ho na zásobník.
  - Pokud je token levá závorka, ulož ji na zásobník.
  - Pokud je token pravá závorka:
    - Dokud na vrcholu zásobníku nebude levá závorka, vybírej operátory ze zásobníku a zapisuj je do výstupní fronty.
    - Vyjmi ze zásobníku levou závorku a „zahoď“ ji.
    - Pokud je zásobník prázdný a nepodařilo se najít levou závorku, jedná se o chybný neuzavřený výraz.
- Jestliže byly zpracovány všechny tokeny ze vstupu:
  - Dokud budou v zásobníku operátory:
    - Jestliže operátor na vrcholu zásobníku je závorka, jedná se o chybný neuzavřený výraz.
    - Vyjmi operátor ze zásobníku a vlož ho do výstupní fronty.
- Konec.

Uvedený algoritmus je zkrácený.

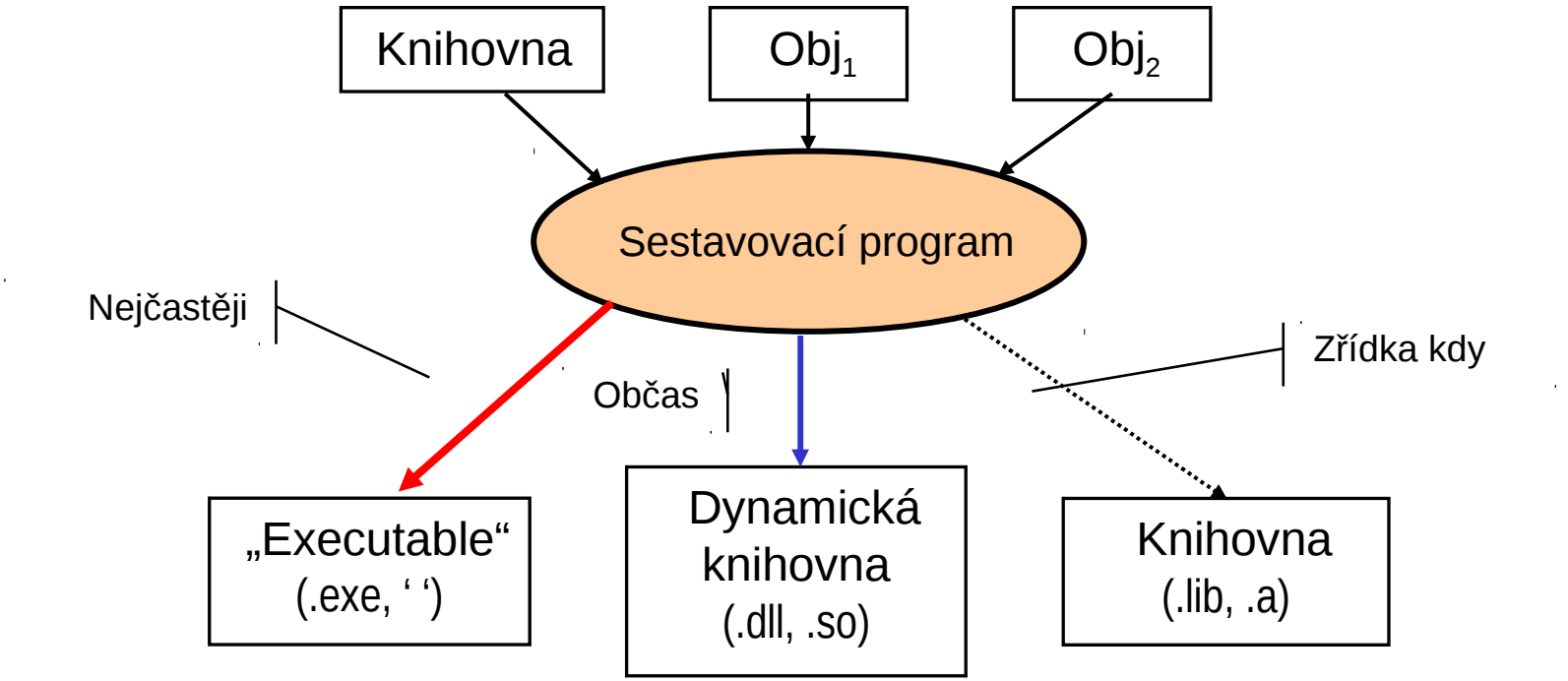
Je vynechána část týkající se funkcí s více argumenty v závorkách oddělenými čárkami, např. zápis `fce(x, y)`.

Neřeší se rovněž priorita a asociativita operátorů bez závorek (odčítání je asociativní zleva, umocňování je asociativní zprava)

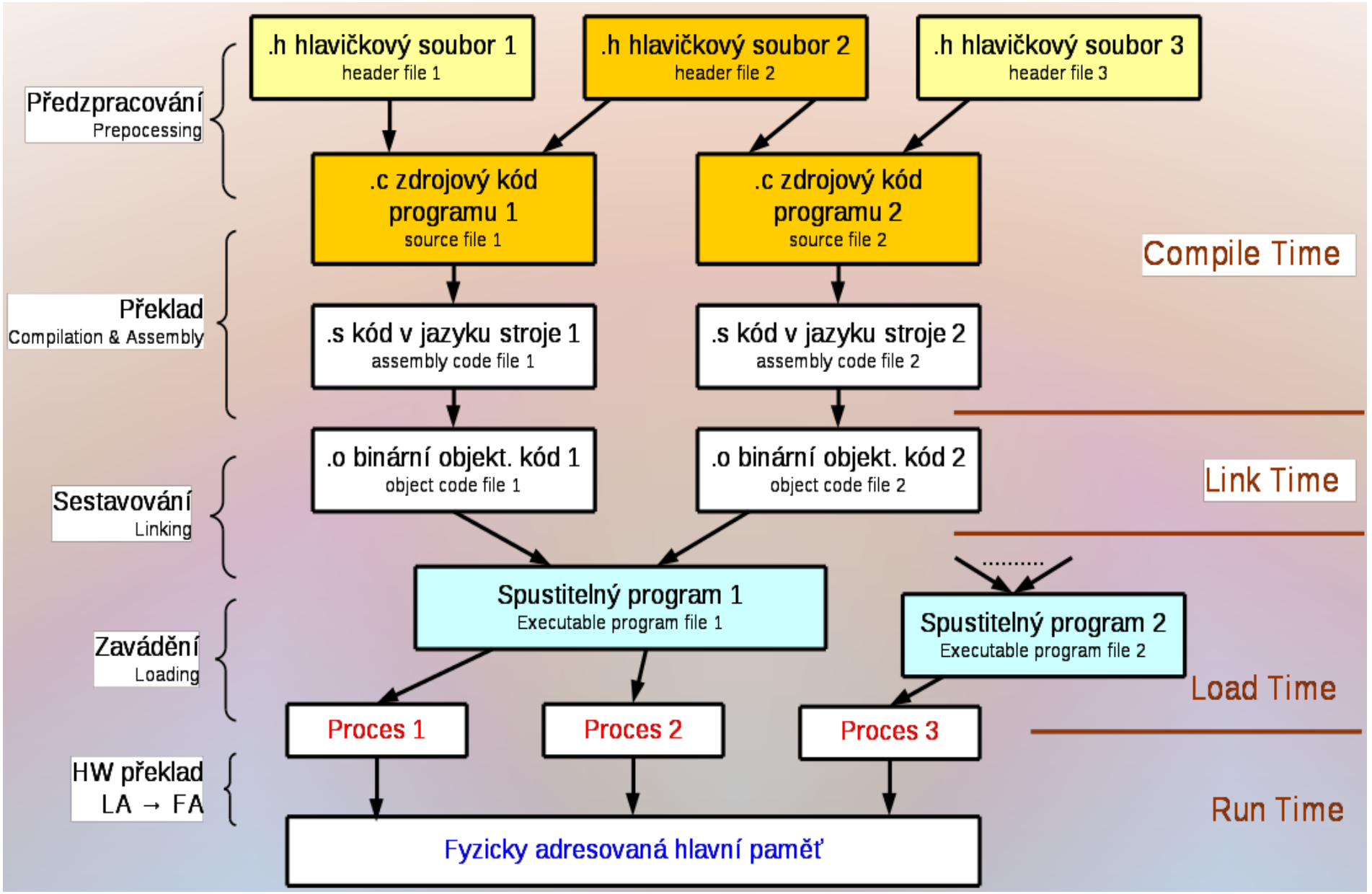
Úplný algoritmus viz např.
http://montcs.bloomu.edu/~bobmon/Information/RPN/infix2rpn.shtml
nebo česky na http://cs.wikipedia.org/wiki/Shunting-yard_(algoritmus)

Binární objektové moduly

- Objektové moduly (*object modules*)
  - Vytvářeny jsou překladači jako binární forma přeloženého programového modulu.
  - Jde o vnitřně strukturovanou kolekci (obvykle pojmenovaných) úseků strojového kódu a dalších informací nutných k následnému zpracování.
  - Objektové moduly jsou vstupem pro sestavovací program (*linker*, někdy též *linkage editor*)



Od zdrojového textu k procesu



Binární objektový modul

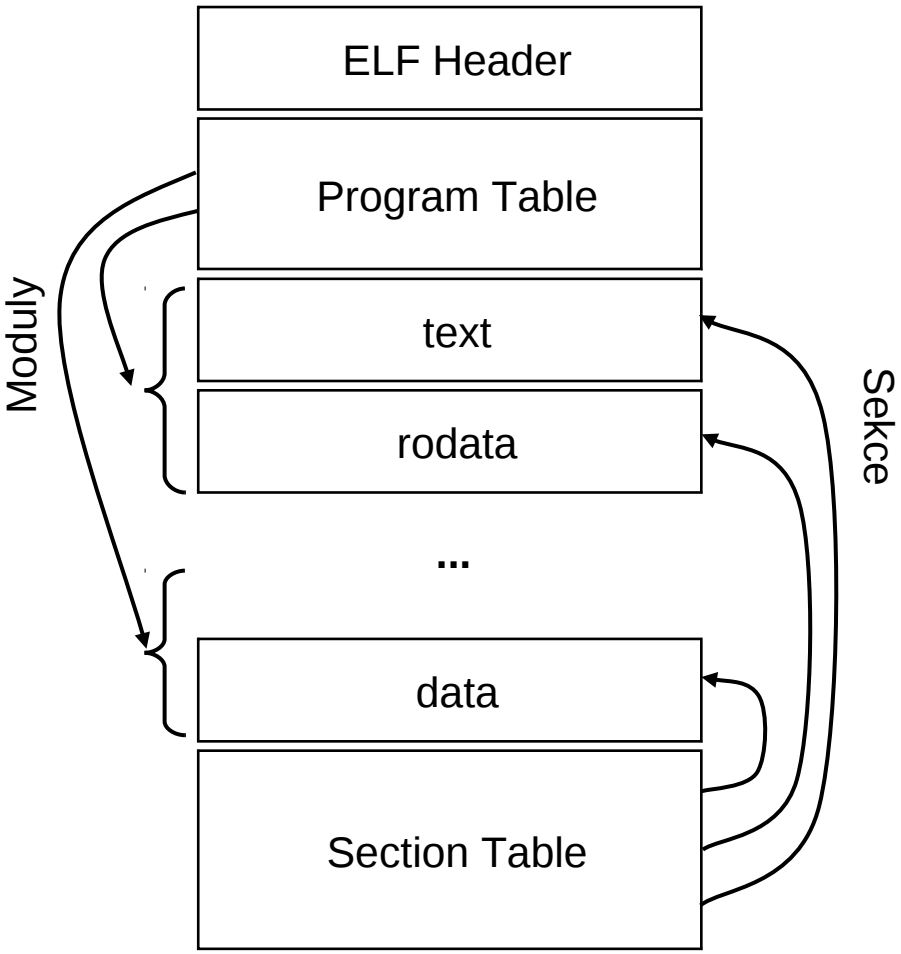
- Každý objektový modul obsahuje sérii sekcí různých typů a vlastností
  - Prakticky všechny formáty objektových modulů obsahují
    - Sekce **`text`** obsahuje strojové instrukce a její vlastností je zpravidla EXEC|ALLOC
    - Sekce **`data`** slouží k alokaci paměťového prostoru pro inicializovaných proměnných, RW|ALLOC
    - Sekce **`bss`** (Block Started by Symbol) popisuje místo v paměti, které netřeba alokovat ve spustitelném souboru, RW
  - Mnohé formáty objektových modulů obsahují navíc
    - Sekce **`rodata`** slouží k alokaci paměťového prostoru konstant, RO|ALLOC
    - Sekci **`symtab`** obsahující tabulku globálních symbolů (➔), kterou používá sestavovací program
    - Sekci **`dynamic`** obsahující informace pro dynamické sestavení
    - Sekci **`dynstr`** obsahující znakové řetězce (jména symbolů pro dynamické sestavení
    - Sekci **`dynsym`** obsahující popisy globálních symbolů pro dynamické sestavení
    - Sekci **`debug`** obsahující informace pro symbolický ladicí program
    - Detaily viz např. http://www.freebsd.org/cgi/man.cgi?query=elf&sektion=5

Formát binárního objektového modulu

- Různé operační systémy používají různé formáty jak objektových modulů tak i spustitelných souborů
- Existuje mnoho různých obecně užívaných konvencí
  - .com, .exe, a .obj
    - formát spustitelných souborů a objektových modulů v MSDOS
  - Portable Executable (PE)
    - formát spustitelných souborů, objektových modulů a dynamických knihoven (.dll) ve MS-Windows. Označení "portable" poukazuje na univerzalitu formátu pro různé HW platformy, na nichž Windows běží.
  - COFF – Common Object File Format
    - formát spustitelných souborů, objektových modulů a dynamických knihoven v systémech na bázi UNIX V
    - Jako první zavedl sekce s explicitní podporou segmentace a virtuální paměti a obsahuje také sekce pro symbolické ladění
  - ELF - Executable and Linkable Format
    - nejpoužívanější formát spustitelných souborů, objektových modulů a dynamických knihoven v moderních implementacích POSIX systémů (Linux, Solaris, FreeBSD, NetBSD, OpenBSD, ...). Je též užíván např. i v PlayStation 2, PlayStation 3 a Symbian OS v9 mobilních telefonů.
    - Velmi obecný formát s podporou mnoha platforem a způsobů práce s virtuální pamětí, včetně volitelné podpory ladění za běhu

Formát ELF

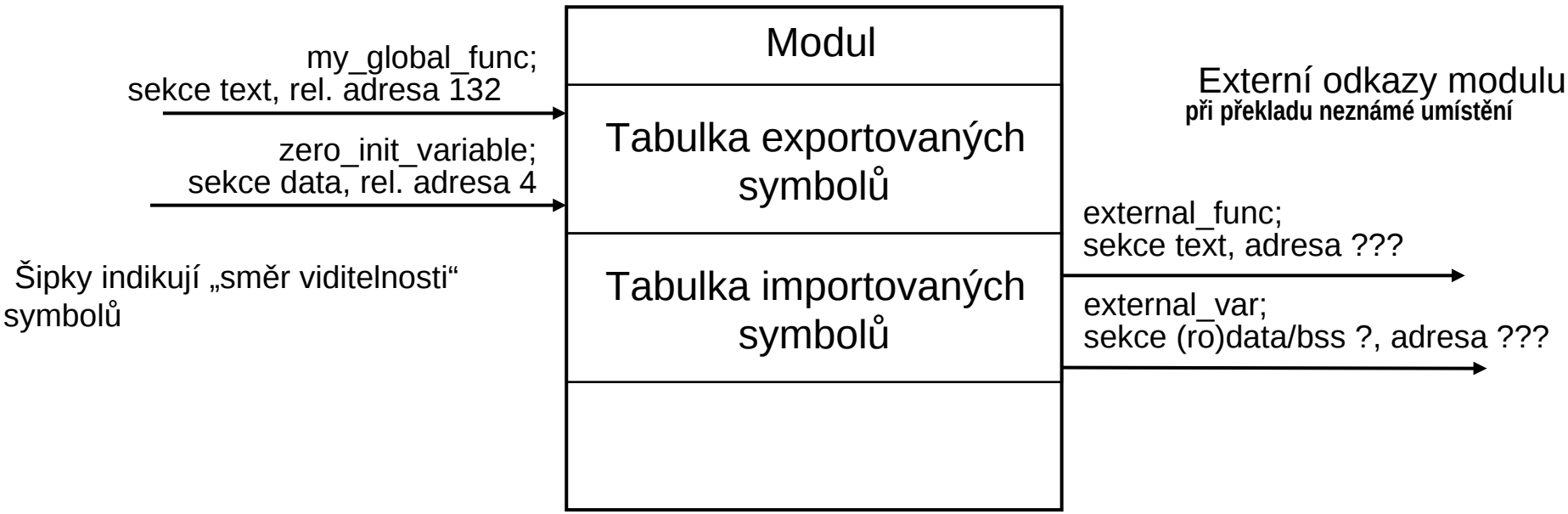
- Formát ELF je shodný pro objektové moduly i pro spustitelné soubory
  - ELF Header obsahuje celkové popisné informace
    - např. identifikace cílového stroje a OS
    - typ souboru (obj vs. exec)
    - počet a velikosti sekci
    - odkaz na tabulku sekcí
    - ...



  - Pro spustitelné soubory je podstatný seznam sekcí i modulů.
  - Pro sestavování musí být moduly popsány svými sekcemi.
  - Sekce jsou příslušných typů a obsahují „strojový kód“ či data
  - Tabulka sekcí popisuje jejich typ, alokační a přístupové informace a další údaje potřebné pro práci sestavovacího či zaváděcího programu

Moduly a globální symboly

- V objektovém modulu jsou (aspoň z hlediska sestavování) potlačeny lokální symboly (např. lokální proměnné uvnitř funkcí – jsou nahrazeny svými adresami, symbolický tvar má smysl jen pro případné ladění)
  - globální symboly slouží pro vazby mezi moduly a jsou 2 typů
    - daným modulem **exportované symboly**. Ty jsou v příslušném modulu plně definovány, je známo jejich jméno a je známa i sekce, v níž se symbol vyskytuje a relativní adresa symbolu vůči počátku sekce.
    - daným modulem požadované **externí** (**importované**) **symboly**, o kterých je známo jen jejich jméno, případně typ sekce, v níž by se symbol měl nacházet (např. pro odlišení, zda symbol představuje jméno funkce či jméno proměnné)

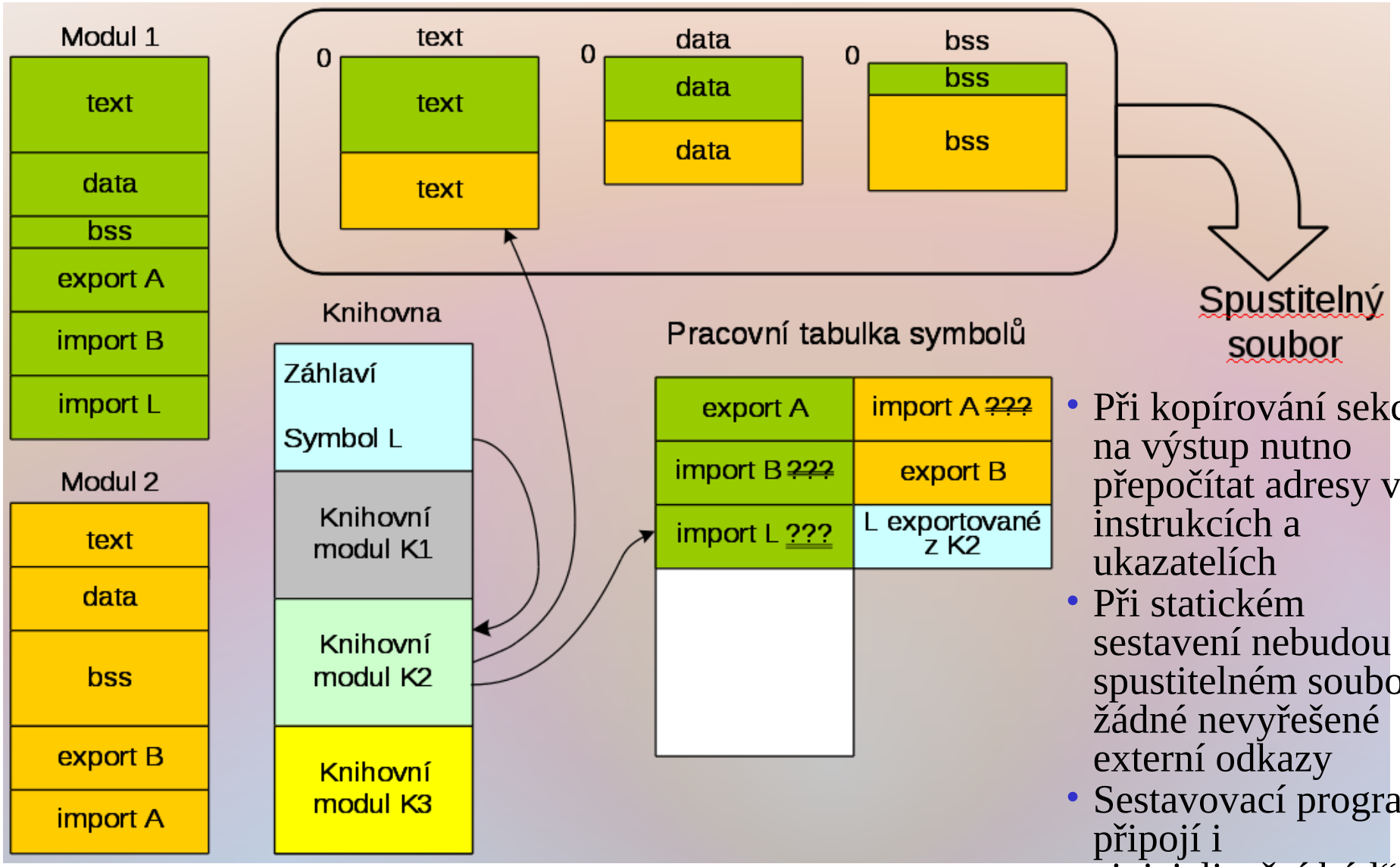


Knihovny

- Knihovny jsou kolekce přemístitelných (objektových) modulů s vnitřní organizací
- Statické knihovny jsou určeny pro zpracování sestavovacím programem
  - Jsou tvořeny záhlavím knihovny (katalogem) a sérií modulů
  - Záhlaví obsahuje rejstřík globálních jmen (symbolů) exportovaných jednotlivými moduly a odkazy na ně.
    - To umožňuje rychlé vyhledání potřebného modulu v knihovně zejména pro statické sestavování (*static build*)
- Dynamické (též sdílené) knihovny jsou složitější
  - Jsou to formálně vlastně spustitelné programy, které „zaregistrují“ v systémových strukturách globální symboly jednotlivých modulů, jimiž je knihovna tvořena, a pak opustí soutěž o procesor.
  - V závislosti na způsobu volání rutin ve sdílené knihovně je mnohdy nutná podpora v jádře OS nebo aspoň ve správě virtuální paměti

Sestavovací program – statické sestavení

- Sestavovací program zpracovává sadu objektových modulů a vytváří soubor se spustitelným programem



- Při kopírování sekcí na výstup nutno přepočítat adresy v instrukcích a ukazatelích
- Při statickém sestavení nebudou ve spustitelném souboru žádné nevyřešené externí odkazy
- Sestavovací program připojí i „inicializační kód“

Sestavovací program – dynamické sestavení

- Sestavovací program
  - pracuje podobně jako při sestavování statickém, avšak na závěr mohou zůstat ve spustitelném souboru nevyřešené odkazy
  - Na místě těchto odkazů připojí sestavovací program malé kousky kódu (zvané ***stub***), které způsobí pozdější vyřešení odkazu
- Existují v zásadě dva přístupy:
  - Vyřešení odkazů při zavádění programu do paměti
    - Zavaděč (*loader*) zkontroluje, zda potřebné dynamické knihovny jsou v paměti (nejsou-li zavede je též), ve virtuální paměti je „namapuje“ tak, aby je zaváděný proces viděl a „stub“ zmodifikuje tak, že odkaz bude vyřešen
  - Vyřešení odkazu skutečně za běhu
    - Vyžaduje se podpora JOS, kdy „stub“ nahrazující nevyřešený odkaz způsobí výjimku a v reakci na ni se provedou akce podobné jako při řešení během zavádění
    - Výhodné z hlediska využití paměti, neboť se nezavádí knihovny, které nebudou potřeba.

Statické versus dynamické sestavení

Jednoduchý příklad

```
hello.c:
#include <stdio.h>
int main() {
        int n = 24;
        printf("%d \tHello, world.\n", n);
}
```

```
hello.o:     file format elf32-i386-freebsd

SYMBOL TABLE:
00000000 *ABS*   00000000 hello.c
00000000 .text   00000000
00000000 .data   00000000
00000000 .bss    00000000
00000000 .rodata         00000000
00000000 .comment        00000000
00000000 .text   00000034 main
00000000 *UND*   00000000 printf
```

File	Size
hello.c	84
hello.asm	472
hello.o	824
hello-static	197970
hello	4846

```
hello.asm:
        .file   "hello.c"
        .section .rodata
.LC0:
        .string "%d \tHello, world.\n"
        .text
        .p2align 4,,15
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $36, %esp
        movl    $24, -8(%ebp)
        movl    -8(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        addl    $36, %esp
        popl    %ecx
        popl    %ebp
        leal    -4(%ecx), %esp
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.2.1 [FreeBSD]"
```

Zavaděč (*loader*)

- Zavaděč je součást JOS, která „rozumí“ spustitelnému souboru
  - V POSIX systémech je to vlastně obsluha služby „**`exec`**“
- Úkoly zavaděče
  - vytvoření „obrazu procesu“ (*memory image*) v odkládacím prostoru na disku
    - a částečně i v hlavní paměti v závislosti na strategii virtualizace
  - vytvoření příslušného deskriptoru procesu (PCB)
  - případné vyřešení nedefinovaných odkazů 🡐
  - sekce ze spustitelného souboru se stávají segmenty procesu
    - pokud správa paměti nepodporuje segmentaci, pak stránkami
      - segmenty jsou obvykle realizovány tak jako tak ve stránkované virtuální paměti
    - segmenty získávají příslušná „práva“ (RW, RO, EXEC, ...)
  - inicializace „registrů procesu“ v PCB
    - např. ukazatel zásobníku a čítač instrukcí
  - předání řízení na vstupní adresu procesu

Poznámky k postupu vzniku procesu

- Zde popsaná problematika je velmi různorodá
  - Popsali jsme jen klasický (byť soudobý) postup
    zdrojový kód → proces
  - Úplný výčet a rozbor možností by vystačil na celý semestr
- K mechanizaci celého postupu slouží obvykle utilita `make`
  - **`make`** má svůj vlastní popisný a (relativně jednoduchý) definiční jazyk
  - zakládá se na časech poslední modifikace souborů
    - např. pokud *file.c* není mladší než *file.o*, netřeba překládat *file.c*
  - např. budování celého operačního systému lze popsat v jediném „Makefile“
- Moderní přístupy s virtuálními stroji používá princip JIC
  - JIC = Just In time Compilation
  - používá např. subsystém .net (nejen ve Windows)
  - principem je, že překlad a sestavení probíhá při zavádění nebo dokonce za běhu

Poznámky k postupu vzniku procesu (pokr.)

- Zavádění (i sestavení) může urychlit použití tzv. PIC
  - PIC = *Position Independent Code*
  - Překladač generuje kód nezávislý na umístění v paměti
    - např. skoky v kódu jsou vždy relativní vůči místu, odkud je skok veden
    - kód je sice obvykle delší, avšak netřeba cokoliv modifikovat při sestavování či zavádění
    - užívá se zejména pro dynamické knihovny
- Objektové programování vyžaduje další podporu jazykově pomocí závislých knihoven
  - např. C++ nebo kompilovaná Java vyžadují podporu tvorby a správy objektů (a *garbage collection*)
- Mnoho optimalizačních postupů při překladu souvisí s architekturou cílového stroje
  - von Neumannova a harwardská architektura mají rozdílné vlastnosti z pohledu optimálního strojového kódu
  - „pipe-lining“ instrukcí a strukturu cache lze lépe využít lépe optimalizovaným kódem
- A mnoho a mnoho dalších „triků“ ...

To je vše.

Otázky?