Překladače – typy, struktura a princip činnosti

Wednesday, May 29, 2013 4:50 PM

Formálně je překladač **zobrazením ze zdrojového jazyka do cílového jazyka**.

Typy

Postup při tvorbě cílového spustitelného programu:

Zdrojový program → [preprocesor] → upravený zdrojový program → [kompilátor]→cílový program v jazyce symbolických instrukcí → [assembler] → relokovatelný strojový kód & zvenku: knihovní soubory a další relokovatelné objektové soubory → [linker/loader] → cílový strojový kód

Preprocesory

Realizují vnořování částí programu do hostitelského jazyka. Např. expandují makra, přidají include <něco.h> apod. Jeho úkolem je posbírat zdrojový program.

Kompilátory

Generují z vyššího programovacího jazyka kód – strojový/symbolický/jiný jazyk (1. Fortran IBM, 50. léta). Assembly language (jazyk symbolických adres) je jednodušší vyplivnout jako výstup a je i jednodušší pro debugování.

Interprety

Namísto přeložení celého programu provádí příkaz za příkazem operace uvedené ve zdrojovém programu nad vstupními daty. Proto jsou interprety obecně pomalejší. Obvykle ale díky spouštění programu příkaz za příkazem lépe diagnostikují chyby než kompilátory.

Java – kombinace kompilace a interpretace

Zdrojový program je nejříve zkompilován do *bytecode* a ten je pak interpretován virtuálním strojem.

Výhoda – bytecode může být zkompilován na jednom stroji a interpretován na jiném. Aby to bylo rychlejší, některé Java kompilátory (*just-in-time* kompilátory) převádí bytecode do strojového jazyka těsně předtím, než proběhne intermediate (vnitřní) program pro zpracování vstupních dat.

Zdrojový program → [Translator] → intermediate program + vstup → [Virtual Machine] → výstup

Assemblery

Překládají z jazyka symbolických instrukcí (JSI) do strojového kódu. Přeložený strojový kód může být buďto *absolutní binární kód* nebo *přenositelný binární kód.*

Hlavní problémy, které řeší, je adresace symbolických jmen a makra.

Linker a loader

Větší programy jsou často kompilované po částech, takže vytvořený relokovatelný strojový kód (= lze jej umístit do libovolného místa v paměti) je potřeba spojit s dalšími relok. objektovými soubory a knihovními soubory. O to se stará **Linker.** Řeší adresy externí paměti, kde kód v jednom souboru může odkazovat na místo v jiném souboru. **Loader** pak narve všechny spustitelné objektové soubory do paměti pro spuštění.

Typy překladačů

Formátory textu

jde o úpravu textu podle požadavků uživatele, např. TeX. Např. syntax highlighting, nebo přeformátování kódu – odsazení apod. Takový překladač, který ze vstupního souboru definovaného určitým jazykem vygeneruje výstup.

Silikonový překladač

Pro návrh integrovaných obvodů. Proměnné nereprezentují místo v paměti, ale logickou proměnnou obvodu. Výstupem je návrh obvodu.

Dávkový překladač

Dávkové zpracování

Inkrementální překladač
Je interaktivní a překládá po úsecích

Křížový překladač
Překládá na jiném procesoru než na kterém se program (přeložený kód) spouští (např. zabudované – embedded – systémy)

Kaskádní překladač
Máme již překlad z jazyka A do jazyka B, chceme ale A→C. Pak vytvoříme kompilátor z jazyka B do jazyka C, pokud je to snazší než vytvořit kompilátor z jazyka A do jazyka C. Jazyk B je vnitřním jazykem, a pokud je to standardní všeobecně používaný jazyk, pak programy v jazyce A budou snadno přenositelné.

Nevýhodou je však, že oba překladače produkují chybové zprávy. Chybové zprávy druhého překladače jsou cizí pro uživatele jazyka A, protože jsou orientovány na jazyk B. Chybová hlášení výpočtu tak budou pomíchaná.

Paralelizující překladač
Zjišťuje nezávislost úseků programu

Optimalizující překladač
Možnosti ovlivnění optimalizace času či paměti programátorem.

Konverzační překladač - interaktivní

Struktura, princip činnosti
Překladače jsou dva druhy: kompilátory a interprety

Struktura Kompilátoru
všechny příkazy překládá najednou, program lze spustit až po ukončení celého překladu (Pascal, C, Fortran, Ada, …)

ANALÝZA: zdrojový program → **lexikální analýza** (lineární), programové symboly → **syntaktická analýza** (hierarchická), derivační strom

SYNTÉZA: derivační strom → **zpracování sémantiky**, program ve vnitřní formě → **optimalizace** (příprava generování), upravený program ve vnitřní formě → **generování kódu**, cílový program

Všechny části spolupracují s pracovními tabulkami překladače. Základní tabulkou kompilátoru i interpretu je **tabulka symbolů.** Obsahuje záznamy o názvech proměnných, jejich typu, rozsahu, názvy procedur společně s věcmi jako počet a typy argumentů, metoda předání jednotlivých argumentů (odkazem nebo hodnotou) a návratový typ.

Výhodou kompilátoru je rychlá exekuce programu.

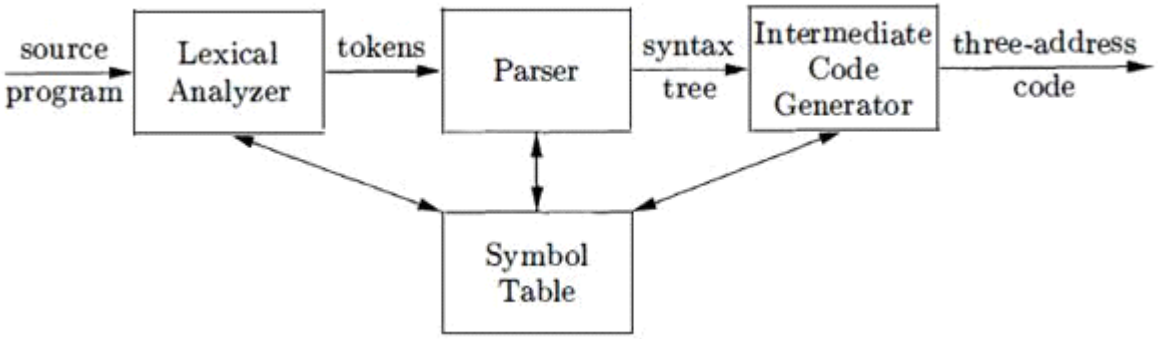
Struktura Interpreta
zpracovává příkazy jednotlivě a každý provede okamžitě po jeho přeložení (Python, Perl, JavaScript, Ruby, …)

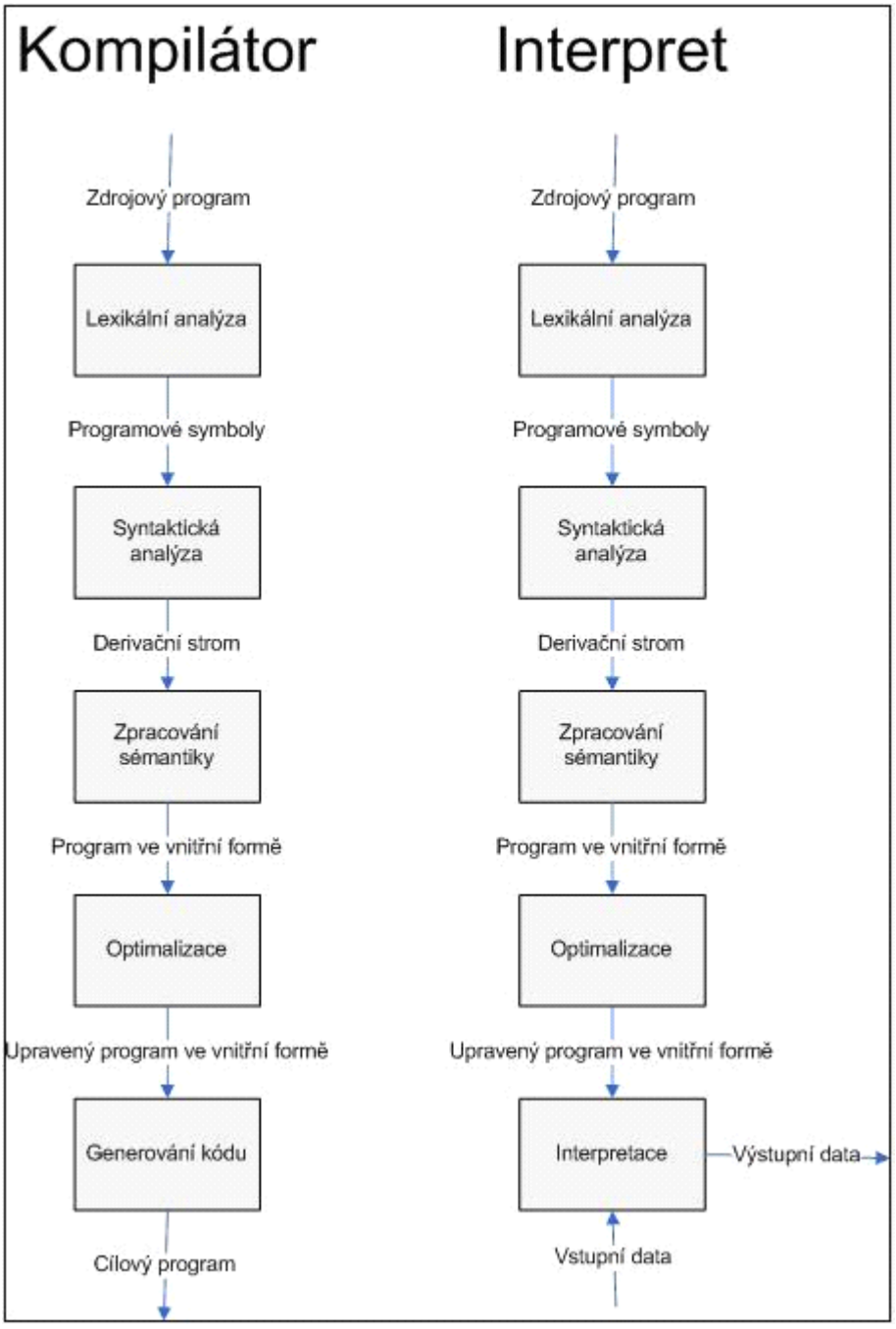
ANALÝZA: zdrojový program → **lexikální analýza** (lineární), programové symboly → **syntaktická analýza** (hierarchická), derivační strom

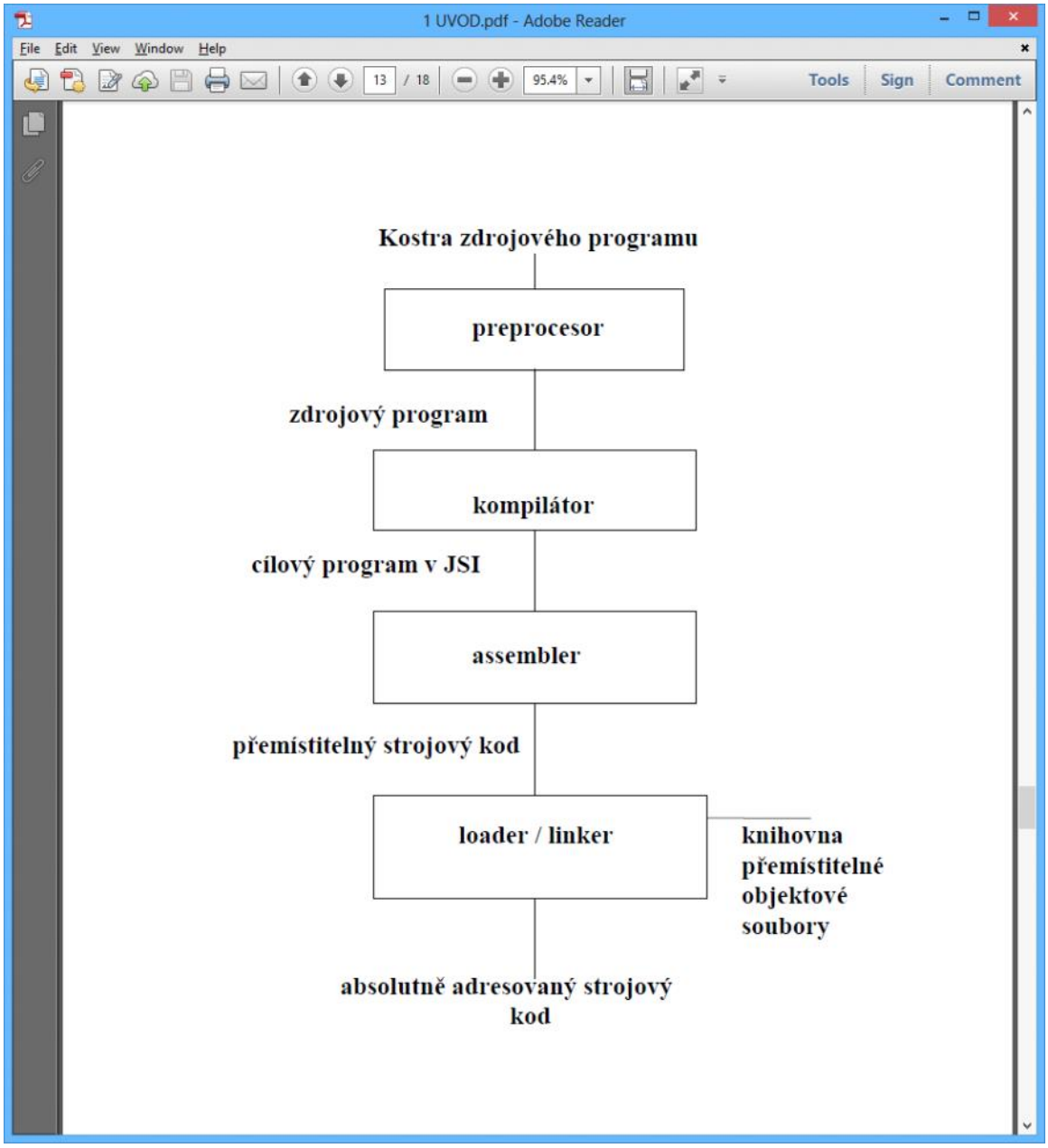
SYNTÉZA: derivační strom → **zpracování sémantiky**, program ve vnitřní formě → **optimalizace** (příprava generování), upravený program ve vnitřní formě → **interpretace**, pracuje se vstupními daty, aby vygeneroval výstupní data

Výhodou interpretu je:

- Eliminace kroků cyklu „editace → překlad → sestavení → exekuce“
- Snazší realizace ladících mechanismů (zachování původních jmen symbolů)


source program
Lexical Analyzer
tokens
Parser
syntax tree
Intermediate Code Generator
three-address code
Symbol Table


Kompilátor
Interpret
Zdrojový program
Lexikální analýza
Programové symboly
Syntaktická analýza
Derivační strom
Zpracování sémantiky
Program ve vnitřní formě
Optimalizace
Upravený program ve vnitřní formě
Generování kódu
Cílový program
Zdrojový program
Lexikální analýza
Programové symboly
Syntaktická analýza
Derivační strom
Zpracování sémantiky
Program ve vnitřní formě
Optimalizace
Upravený program ve vnitřní formě
Interpretace
Výstupní data
Vstupní data

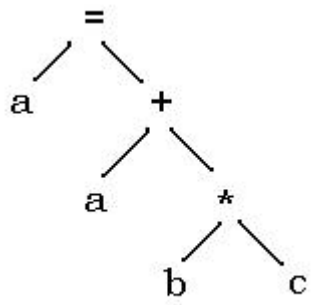


Lexikální analýza

zdrojový kód vstupuje do procesu překladu jako posloupnost znaků. Tato posloupnost se čte lineárně zleva doprava a sestavují se z ní *lexikální symboly* jako konstanty, identifikátory, klíčová slova nebo operátory. Je založena na regulárních gramatikách. Výsledkem je posloupnost symbolů, např. je na vstupu rozeznáno klíčové slovo begin a do posloupnosti lexikálních symbolů bude zařazen nový symbol reprezentující právě toto klíčové slovo. Tyto symboly jsou programem snadno použitelné a dále zpracovatelné. V této fázi se odstraňují veškeré komentáře.

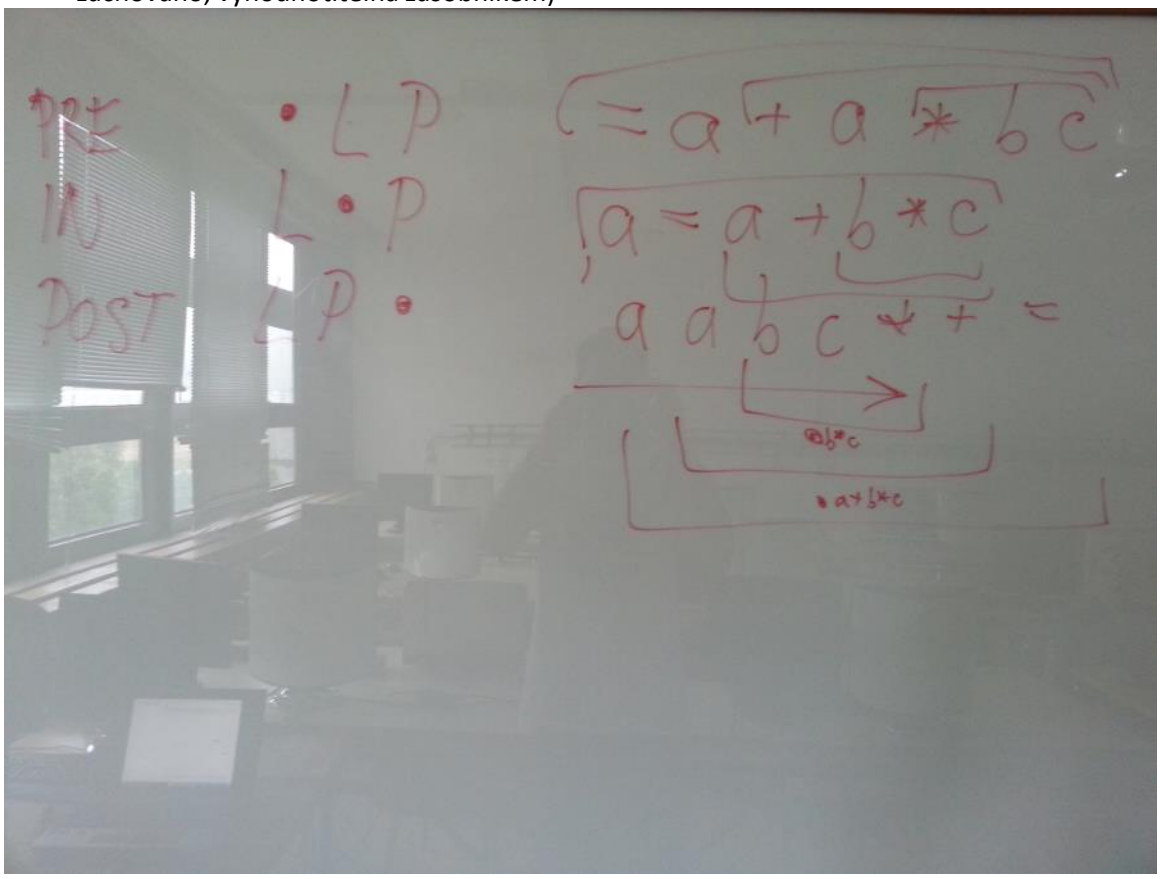
Syntaktická analýza

Z posloupnosti lexikálních symbolů se vytvářejí hierarchicky zanořené struktury (vnitřní jazyk překladače), které mají jako celek svůj vlastní význam, např. výrazy, příkazy, deklarace nebo program. Programy jsou psány většinou v infixové notaci (a=a+b*c) => analyzujeme a vytváříme hierarchické uspořádání derivačního stromu:



Notace vnitřního jazyka překladače:

- Prefixová (nemá závorky, operátory bezprostředně *předchází* operandy a pořadí operandů je zachováno)
- Infixová
- Postfixová (nemá závorky, operátory bezprostředně *následují* operandy a pořadí operandů je zachováno, vyhodnotitelná zásobníkem)



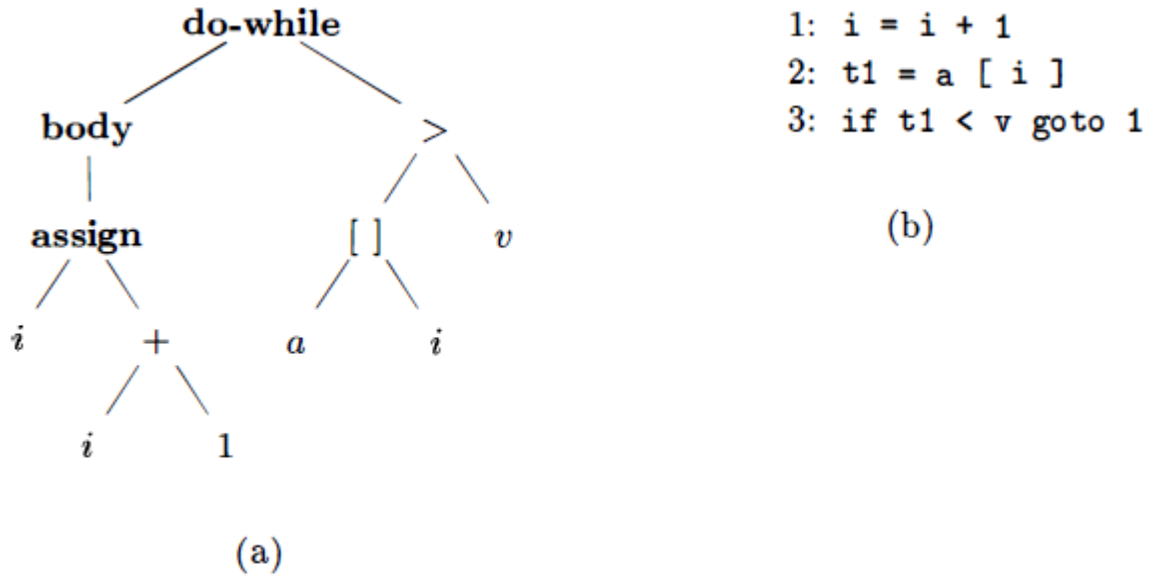


Figure 2.4: Intermediate code for "`do i = i + 1; while ( a[i] < v);`"

Sémantická analýza

Provádějí se některé kontroly, zajišťující správnost programu z hlediska vazeb, které nelze provádět v rámci syntaktické analýzy (některé konstrukty nejdou popsat bezkontextovou gramatikou, třeba např. kontrola deklarací, typová kontrola, kontrola, jestli index pole je integer apod.).

Typická reprezentace programu ve *vnitřní formě* (intermediate code) je sekvence trojic nebo čtveřic (3- nebo 4-adresových instrukcí)

Optimalizace

Optimalizátor kódu zajišťuje, aby se používalo co nejméně pomocných proměnných pro mezivýpočty, aby se v cyklu zbytečně několikrát nevyhodnocoval tentýž výraz, jestliže hodnota jeho prvků zůstává bez změny a vyhodnocení stačí provést jednou před cyklem, apod. Optimalizací prochází program obvykle v intermediálním tvaru – intermediální kód je již podobný cílovému programu, má však strukturu vhodnější pro optimalizaci. Může to být zápis podobný assembleru nebo třeba dynamická struktura (dynamický seznam stromů představujících jednotlivé příkazy).

Generování kódu

poslední fází překladače je generování cílového kódu, což je obvykle přemístitelný kód nebo program jazyka asembleru. Všem proměnným použitým v programu se přidělí místo v paměti. Potom se instrukce mezikódu překládají do posloupnosti strojových instrukcí, které provádějí stejnou činnost.

Vícefázový a víceprůchodový překladač

Fáze = logicky dekomponovaná část (může obsahovat více průchodů, např. optimalizace)

Průchod = čtení vstupního řetězce, zpracování, zápis výstupního řetězce – může obsahovat více fází

Jednoprůchodový překladač = všechny fáze probíhají v rámci jediného čtení zdrojového textu programu

- Omezená možnost kontextových kontrol
- Omezená možnost optimalizace
- Lepší možnosti zpracování chyb a ladění (tedy dobré pro výuku)

Na strukturu překladače mají vliv:

- Vlastnosti zdrojového a cílového jazyka
- Vlastnosti hostitelského počítače

- Rychlost/velikost překladače
- Rychlost/velikost cílového kódu
- Ladicí schopnosti (detekce chyb, zotavení)
- Velikost projektu, prostředky, termíny

Testování a údržba překladače

Díky formální specifikaci jazyka je možné automatické provádění testů

Systematického testování lze dosáhnout regresními testy, což je sada testů doplňovaná o testy na odhalené chyby. Po každé změně v překladači se provedou všechny testy a jejich výstupy se porovnají s předešlými.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>