KOMPILÁTORY: Úvod

Jana Dvořáková
dvorakova@dcs.fmph.uniba.sk

Literatúra

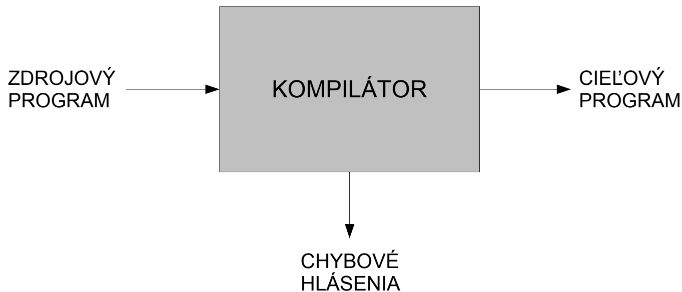
- Green Dragon Book
  Aho, Ullman: *Compiler Design*. Addison Wesley 1977
- Red Dragon Book *
  Aho, Sethi, Ullman: *Compilers: Principles, Techniques and Tools*. Addison Wesley 1986
- Purple Dragon Book *
  Aho, Lam, Sethi, Ullman: *Compilers: Principles, Techniques and Tools (2nd edition)*. Addison Wesley 2006
- Aho, Ullman: *The Theory of Parsing, Translation and Compiling*.
  Vol I. Parsing. Prentice Hall 1972
  Vol II. Compiling. Englewood Cliffs N.J. 1973
- Prednášky:
  `http://foja.dcs.fmph.uniba.sk/kompilatory`

Predpoklady

1. Formálne jazyky a automaty
2. Praktická znalosť nejakého programovacieho jazyka
3. Assembler

Čo je to kompilátor?

Prvá predstava



Čo je to kompilátor?

O niečo formálnejšie

- Majme vstupný jazyk L_{in} generovaný gramatikou G_{in}
- Ďalej majme výstupný jazyk L_{out} generovaný gramatikou G_{out}
- Kompilátor je zobrazenie $L_{in} \rightarrow L_{out}$, kde $\forall\, w_{in} \in L_{in}$ $\exists\, w_{out} \in L_{out}$. Pre $w_{in} \notin L_{in}$ zobrazenie neexistuje.

Netypické kompilátory

- Dekompilátory
  - Prekladajú z nižšieho programovacieho jazyka do vyššieho
- Prekladače jazykov (source-to-source compilers)
  - Prekladajú medzi vyššími programovacími jazykmi
- Postupné kompilátory (stage compilers)
  - Kompilujú do jazyka pre abstraktný stroj (Java Bytecode, CIL v Microsot .NET)
- JIT (just-in-time) kompilátory
  - Kompilácia do strojového kódu prebieha tesne pred vykonaním kódu (CIL v Microsot .NET)

- Interpretery = alternatíva ku kompilátorom (Java, skriptovacie jazyky)

Prečo sa zaoberať kompilátormi?

Historická perspektíva

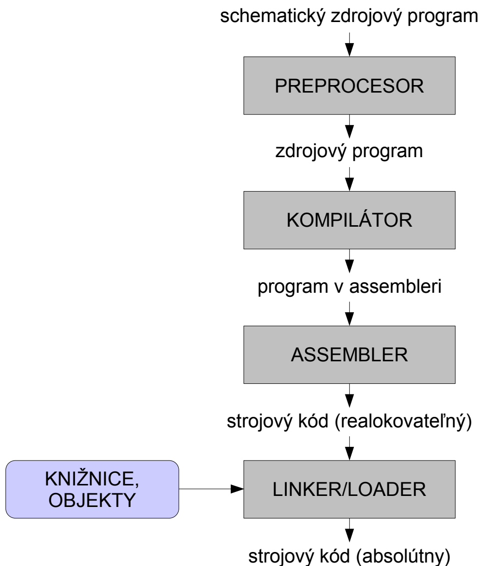
- 50te roky - prvé kompilátory
- Kompilátory považované za programy náročné na vývoj
  - Vývoj kompilátoru pre FORTRAN (1957) trval veľkému pracovnému tímu 3 roky
- Neskôr nájdené efektívne implementačné techniky
  - V súčastnosti dokáže vytvoriť jednoduchý kompilátor aj študent ako semestrálny projekt
- Techniky kompilátorov majú široké využitie, je užitočné si ich osvojiť

Prečo sa zaoberať kompilátormi?

Príklady využitia techník kompilátorov

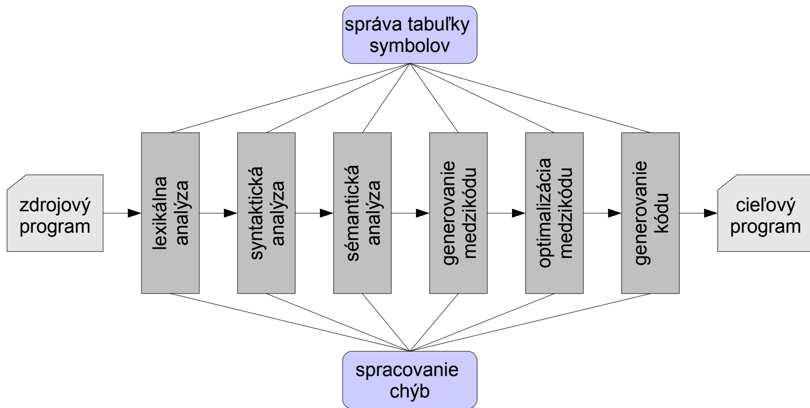
- Štruktúrované alebo syntax-highlighting editory
- Pretty printery
- Statické kontroly programu (LINT)
- Interpretery
- Formátory textu (LaTeX)
- Interpretery dotazovacích jazykov (SQL)
- Spracovanie štruktúrovaných dokumentov (XML)

Spracovanie programu

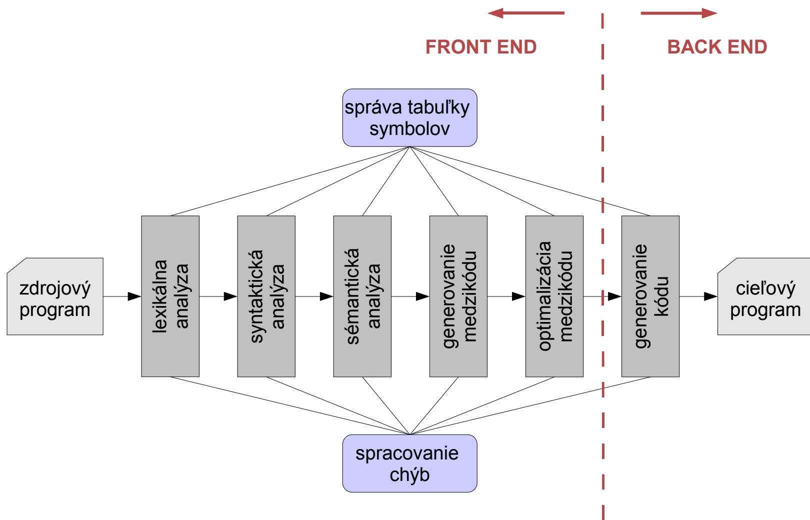


Štruktúra kompilátora

Fázy kompilácie



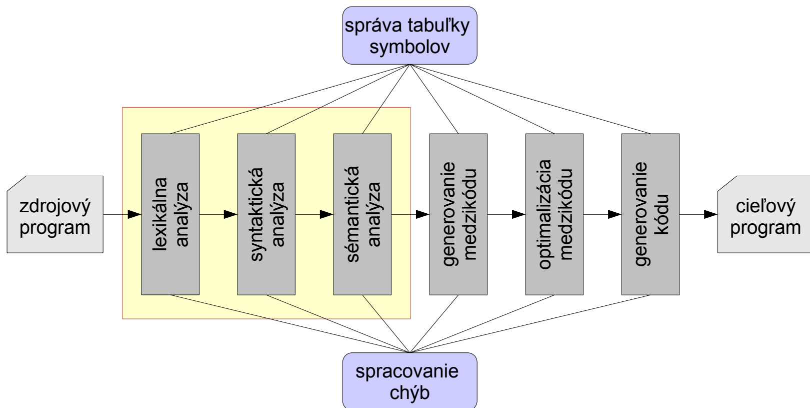
Fázy kompilácie



- Front end: fázy závislé na zdrojovom jazyku
- Back end: fázy závislé na cieľovom jazyku (procesore)
- Rozdeľuje ich medzikód - prechodná reprezentácia zdrojového programu
- Výhody:
  - Rôzne back-endy pre rovnaký vstupný jazyk
  - Rôzne front-endy pre rovnaký výstupný jazyk
  - Strojovo nezávislá optimalizácia medzikódu - HLO
- Existujú kompilátory podporujúce rôzne jazyky a rôzne procesory
  - Napr. GNU Compiler Collection
  - spoločný medzikód

Fázy kompilácie

Analýza



Lexikálna analýza

- Znaky zdrojového programu sú načítané a zoskupené do *tokenov*

Tokeny

= reťazce so spoločným významom, tvoria výstup lexikálnej analýzy

- Napr. := (znak priradenia), begin (klúčove slovo begin), id (identifikátor),..
- Definované pomocou regulárnych výrazov
- Rozpoznávanie tokenov = rozpoznávanie regulárneho jazyka

vstup: `dráha := počiatok + čas * 60`
výstup: `id := id + id * 60`

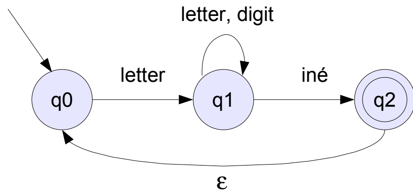
lexéma	**token**	
`dráha`	`id`	(identifikátor)
`:=`	`:=`	(symbol priradenia)
`počiatok`	`id`	(identifikátor)
`+`	`+`	(operátor sčítania)
`čas`	`id`	(identifikátor)
`*`	`*`	(operátor násobenia)
`60`	`60`	(konštanta)

Rozpoznávanie identifikátorov:

1. Regulárne výrazy

```
id      →  letter(letter|digit)*
letter  →  (A|...|Z|a|...|z)
digit   →  (0|...|9)
```

2. Konečný automat



Syntaktická analýza

- Tokeny sú zoskupené do stromovej štruktúry
  - Jednotlivé uzly predstavujú konštrukcie vstupného jazyka
  - Napr. program, príkaz, priradenie, výraz,..
- Štruktúra je popísaná bezkontextovou gramatikou
  - Postačuje pre syntax väčšiny programovacích jazykov
- Generuje sa strom odvodenia (resp. syntaktický strom) daného vstupu
- Problém s nejednoznačnými gramatikami

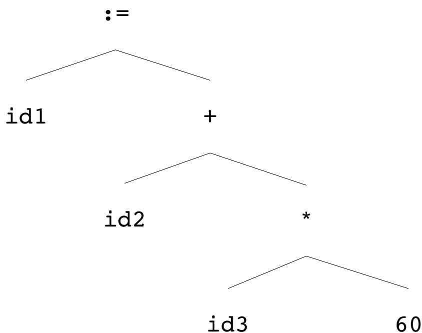
Rozpoznávanie priradenia:

- Pravidlá bezkontextovej gramatiky

$$
\begin{array}{lcl}
\textit{assignment} & \rightarrow & \textbf{id} := \textit{expr} \\
\textit{expr} & \rightarrow & \textit{expr} + \textit{term} \mid \textit{expr} - \textit{term} \mid \textit{term} \\
\textit{term} & \rightarrow & \textit{term} * \textit{factor} \mid \textit{term} / \textit{factor} \mid \textit{factor} \\
\textit{factor} & \rightarrow & \textbf{digit} \mid \textbf{id} \mid (\textit{expr})
\end{array}
$$

vstup: `id1 := id2 + id3 * 60`

výstup: syntaktický strom



- Algoritmy CYK (Cocke-Younger-Kasami), Earley: $O(n^3)$
- Pre podmnožiny $\mathcal{L}_{\mathcal{CF}}$ existujú aj algoritmy v čase $O(n)$:
  - Zhora-nadol *(TOP-DOWN)*:
    - Syntaktický strom sa generuje od koreňa k listom
    - Konštruuje sa *ľavé* krajné odvodenie pre daný vstup
    - Zvyčajne riešené *rekurzívnym zostupom*
    - LL(1) gramatiky
  - Zdola-nahor *(BOTTOM-UP)*:
    - Syntaktický strom sa generuje od listov ku koreňu
    - Konštruuje sa *pravé* krajné odvodenie pre daný vstup
    - LR(1) gramatiky = väčšia podmnožina $\mathcal{L}_{\mathcal{CF}}$ ako LL(1)

Lexikálna vs. syntaktická analýza

- Hranica medzi LA a SA nie je fixná
- LA by mohla byť súčasťou SA, ale rozdelenie je výhodné:
  - Zjednodušenie analýzy (rozdelenie na podúlohy)
  - Väčšia možnosť špecializácie
- Čo dať do LA a čo do SA?
  - LA: konštrukcie definované bez rekurzie (identifikátory,..)
  - SA: konštrukcie definované pomocou rekurzie (príkazy, výrazy,..)

Syntaxou riadený preklad

- Prepojuje syntaktickú analýzu s ďalšími fázami kompilácie
- Pomáha sceliť front-end kompilátora

- K pravidlám gramatiky sú priradené sémantické akcie
  - Sémantická analýza, generovanie medzikódu
- Pri rozpoznaní pravidla sa vykonajú príslušné akcie

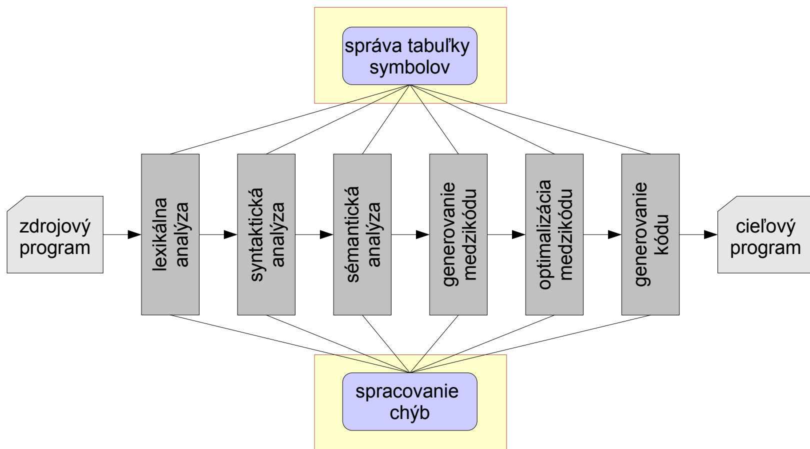
- Notácia:
  1. Syntaxou riadená definícia
  2. Prekladová schéma

Sémantická analýza

- "Všetko, čo sa nedá vyjadriť bezkontextovým jazykom"
- Kontrola sémantických chýb v zdrojovom programe
- Zozbieranie informácie o typoch pre ďalšie fázy (syntézu)
- Dôležitou úlohou je *typová kontrola*, napr.:
  - Pre operátory kontrola typov operandov
  - Pre funkcie kontrola počtu a typov argumentov/návratovej hodnoty
- Niektoré typové konverzie sú automatické (`int` na `real` pri aritmetických operátoroch)

Fázy kompilácie

Tabuľka symbolov a spracovanie chýb



Tabuľka symbolov

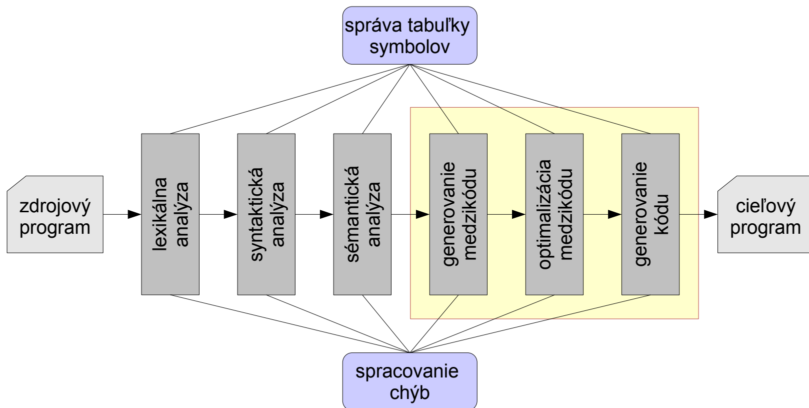
- Dátova štruktúra, kde sa počas analýzy ukladajú informácie o identifikátoroch
- Obsahuje záznam pre každý identifikátor s poľami pre jeho atribúty
    - Alokovaná pamäť
    - Typ
    - Rozsah platnosti
    - Pri menách procedúr aj počet a typ parametrov, spôsob predávania parametrov, typ návratovej hodnoty
- Požiadavky: rýchle vyhľadávanie záznamov, rýchle pridávanie a modifikovanie záznamov
- Rôzne atribúty sú pridávané počas rôznych fáz

Spracovanie chýb

- Chyby sa môžu vyskytnúť počas ľubovolnej fázy
  - Lex. analýza: nespracované znaky na vstupe netvoria žiadny token
  - Synt. analýza: postupnosť tokenov nespĺňa štrukturálne pravidlá
  - Sém. analýza: syntaktická konštrukcia nemá význam vzhľadom na danú operáciu
- Každá fáza by mala byť schopná sa z chýb rozumne zotaviť

Fázy kompilácie

Syntéza



Generovanie medzikódu

- Syntaxou riadený preklad - vytvára sa syntaktický strom = zjednodušený strom odvodenia
- *Medzikód* = reprezentácia syntaktického stromu
  - Jednoducho generovateľný
  - Jednoduchý preložiteľný do cieľového jazyka
- Zvyčajne reprezentovaný lineárne ako trojadresový kód
  - $x := y \ op \ z$
  - Implementovaný ako záznam so štyrmi položkami: $op, arg1, arg2, res$

Optimalizácia medzikódu

- Transformácie medzikódu
  - Optimalizácia na rýchlosť
  - Optimalizácia na veľkosť kódu
  - (Napr. eliminácia spoločných podvýrazov, eliminácia mŕtveho kódu, optimalizácia cyklov..)
- Nad medzikódom sa vykonáva *vysokoúrovňová optimalizácia* (HLO)
- V rôznych kompilátoroch rôzna miera optimalizácie
  - Optimalizačné kompilátory - fáza optimalizácie zaberá podstatnú časť práce kompilátora

Generovanie kódu

- Priradenie miest v pamäti pre jednotlivé premenné
- Preklad inštrukcií medzikódu do postupností inštrukcií cieľového jazyka (assembler, strojový kód)
  - Kľúčové je priradenie premenných do registrov
  - Základnou technikou je ¨template matching¨ - príkazy medzikódu sú nahradené postupnosťami strojových inštrukcií (= templates)

Príklad

```
draha := pociatok + cas * 60
```

Lexikálny analyzátor

```
id1 := id2 + id3 * 60
```

Syntaktický analyzátor

```
        :=
   id1       +
        id2       *
             id3      60
```

Sémantický analyzátor

```
        :=
   id1       +
        id2       *
             id3  inttoreal
                      |
                      60
```

Generátor medzikódu

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Optimalizátor medzikódu

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Generátor kódu

```
MOVF id3, R2
MULF #60.0 R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Nástroje pre tvorbu kompilátorov

- Generátory scannerov
  - Generujú lexikálny analyzátor
  - Vstupom je obvykle regulárna gramatika
  - Flex
- Generátory parserov
  - Generujú syntaktický analyzátor
  - Vstupom je obvykle bezkontextová gramatika
  - Bison, ANTLR
- Generátory generátorov kódu
  - Generujú preklad inštrukcií medzikódu do cieľového kódu
  - Využíva sa ¨template matching¨