

# Formální jazyky a překladače

## **Přednášky:**

- Typy překladačů, základní struktura překladače
- Regulární gramatiky, konečné automaty a jejich využití v lexikální analýze
- Úvod do syntaktické analýzy, metoda rekurzivního sestupu
- Překlad příkazů
- Zpracování deklarací
- Přidělování paměti
- Interpretační zpracování
- Generování cílového kódu
- Vlastnosti bezkontextových gramatik
- Deterministická analýza shora dolů
- LL(1) transformace
- Deterministická analýza zdola nahoru
- Formální překlady

## **Cvičení:**

- Opakování teoretického základu
- Generátor lexikální analýzy - LEX
- Jazyk PL0 a jeho překladač
- Rozšíření konstrukcí jazyka PL0, zadání individuálních úloh
- Příklady LL gramatik
- Příklady LR gramatik, ukázky práce s generátorem YACC

## **Zápočet:**

Udělen na základě referátu a předvedení zadané modifikace překladače PL0

## **Zkouška:**

Písemná forma - 2 příklady (s použitím vlastní literatury) a otázky (bez použití literatury). Výsledky budou spolu s termíny pro reklamaci, zápis do indexu či ústní přezkoušení zveřejněny na webu a nástěnce. Po uplynutí termínu bude výsledek zapsán do databáze známek i v případě, že student nepředložil index k zápisu.

**Literatura:** web stránky předmětu FJP

Melichar, Češka, Ježek, Rychta: Konstrukce překladačů (ČVUT)

Mogensen: Basics of Compiler Design

([http://www.diku.dk/~torbenm/Basics/basics\\_lulu2.pdf](http://www.diku.dk/~torbenm/Basics/basics_lulu2.pdf))

Melichar: Jazyky a překlady (ČVUT)

Molnár a kol.: Gramatiky a jazyky (Alfa)

Doporučená (v knihovně): Aho, Sethi, Ullman: Compilers Principles Technics and Tools  
Appel A.W.: Modern Compiler Implementation in Java

## Formální jazyky a překladače - organizace

FJP je šestikreditový předmět doporučený pro 2. ročník *Informatika a výpočetní technika*. K získání zkoušky je zapotřebí splnit požadavky cvičení a napsat zkouškový test. Celkové hodnocení se snadno zjistí z bodového zisku a následující převodní tabulky:

Více než 82 b.	výborně
65 – 82 b.	velmi dobře
51 – 64 b.	dobře
0 – 50 b.	nevyhověl

Na některých cvičeních se zadává samostatné vyřešení příkladu do příštího cvičení; správné řešení je honorováno polovinou bodu. V rámci cvičení vypracovávají studenti semestrální práci. Hodnocení lehké semestrální práce je až 30 bodů, těžké až 40 bodů. Termín odevzdání práce je do 10. 1. Za každý den prodlení se automaticky strhává 1 bod. Ze semestrální práce je zapotřebí získat alespoň 20 bodů.

Zkouška probíhá písemně. Maximální hodnocení je 60 bodů. Ze zkouškového testu musí student získat alespoň 30 bodů. V polovině semestru mají studenti možnost napsat dobrovolně jednoduchý test z doposud probrané látky.

**Stránky předmětu:**

[Portál ZČU](#) > [Courseware](#) > [Předměty po fakultách](#) > [Fakulta aplikovaných věd](#) > [Katedra informatiky a výpočetní techniky](#) > [FJP](#) > O předmětu

## Využití teorie překladačů a formálních jazyků

- **Assemblery** překlad z JSI. Hlavní problém = adresace symbolických jmen, makra
- **Kompilátory** generují kód (strojový / symbolický / jiný jazyk)
- **Interprety** provádí překlad i exekuci programu převedeného do vhodné formy
- **Strukturní editory** napovídají možné tvary konstrukcí programů, či strukturovaných textů
- **Pretty printers** provádí úpravu struktury výpisů
- **Statické odladovače** vyhledávání chyb bez exekuce programu
- **Reverzní překladače** převádí strojový kód do JSI / vyššího jazyka
- **Formátory textu** překladače pro sazbu textu (Tex→DVI)
- **Silikonové překladače** pro návrh integrovaných obvodů.  
Proměnné nerepresentují místo v paměti, ale log. proměnnou obvodu. Výstupem je návrh obvodu.
- **Příkazové interprety** pro administraci OS / sítí (viz shell Unixu)
- **Dotazovací interprety** analýza a překlad příkazů a podmínek dotazů a příkazů DB jazyků
- **Preprocesory** realizují vnořování částí programu do hostitelského jazyka (expandují makra, přidají include <něco.h> soubory apod.)
- **Analyzátoři textu** kontroly pravopisu, práce s dokumenty, vyhledávání, indexace, zpracování XML.

**Znalost principů překladače patří k základním disciplínám informatiky. Programátorům umožní vytváření lepších programů v konkrétních jazycích, pochopit výhody a úskalí konkrétních programových struktur a lépe porozumět příčinám chyb, které překladač hlásí.**

Formálně je překladač zobrazením:

**Překladač: zdrojový jazyk**  $\longrightarrow$  **cílový jazyk**

**Činnost assembleru:** **JSI**  $\longrightarrow$  **strojový kód**  
absolutní binární kód /přemístitelný binární kód

**Činnost kompilátoru: vyšší progr. Jazyk**  $\longrightarrow$  **strojový kód**  
Pozn.: První překladač – Fortran IBM (Backus 1957)  
pracnost 18 člověkoroků -ad hoc technologie

**Činnost interpretu: vyšší progr. jazyk**  $\Longrightarrow$  **výsledky data**

**Dávkový překladač** batchové zpracování

**Konverzační překladač** interaktivní

**Inkrementální překladač** interaktivní + překládá po úsecích  
(př. Basic překlad po řádcích)

**Křížový překladač** překlad na jiném procesoru než  
exekece (viz zabudované systémy)

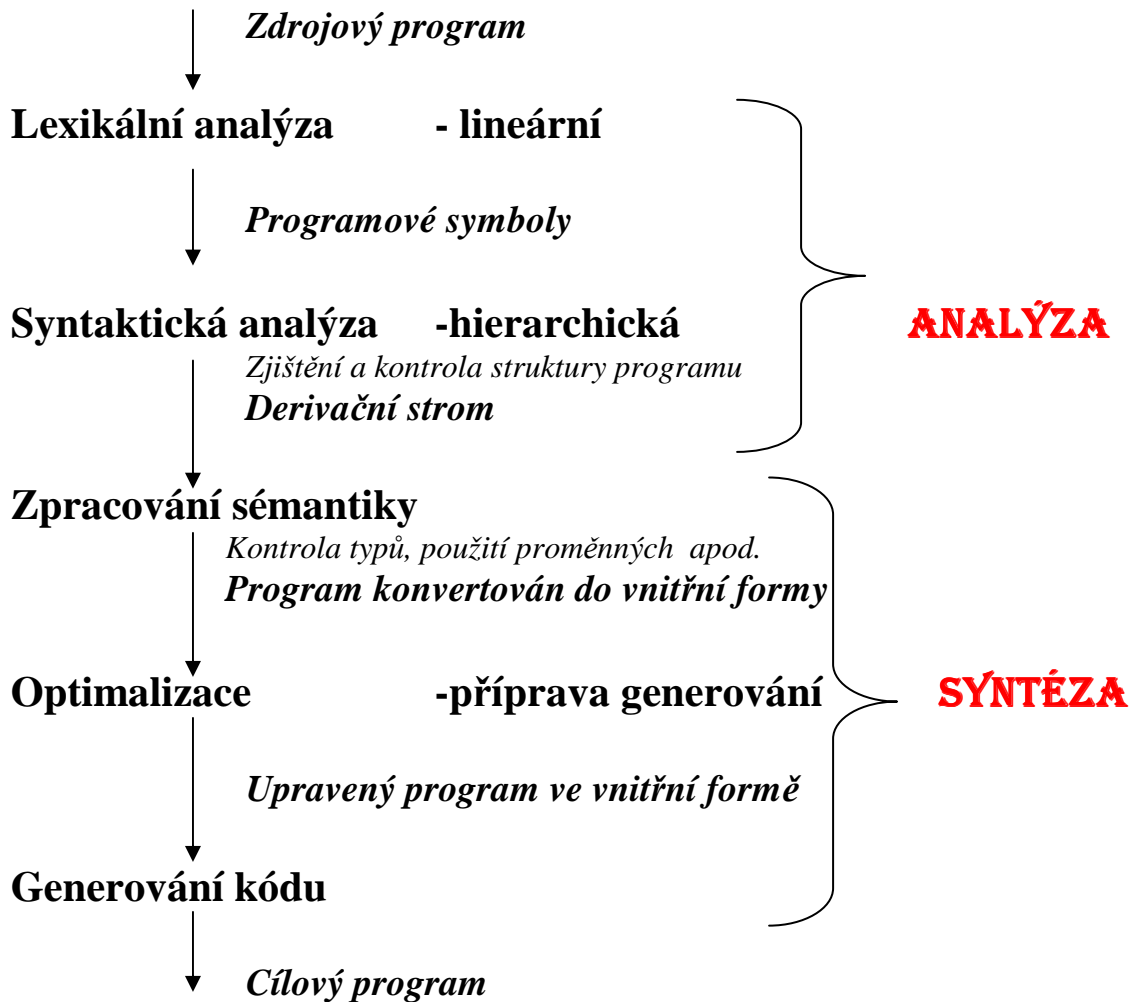
**Kaskádní překladač** máme již  $A \rightarrow B$ , ale chceme  $A \rightarrow C$ ,  
uděláme  $B \rightarrow C$ . Kdy se to vyplatí?  
Komplikace - chybová hlášení výpočtu  
jsou pomíchaná

**Optimalizující překladač** (možnost ovlivnění optimalizace času /  
paměti programátorem)

**Paralelizující překladač**  $\approx$  zjišťuje nezávislost úseků programu

## Hlavní části překladače

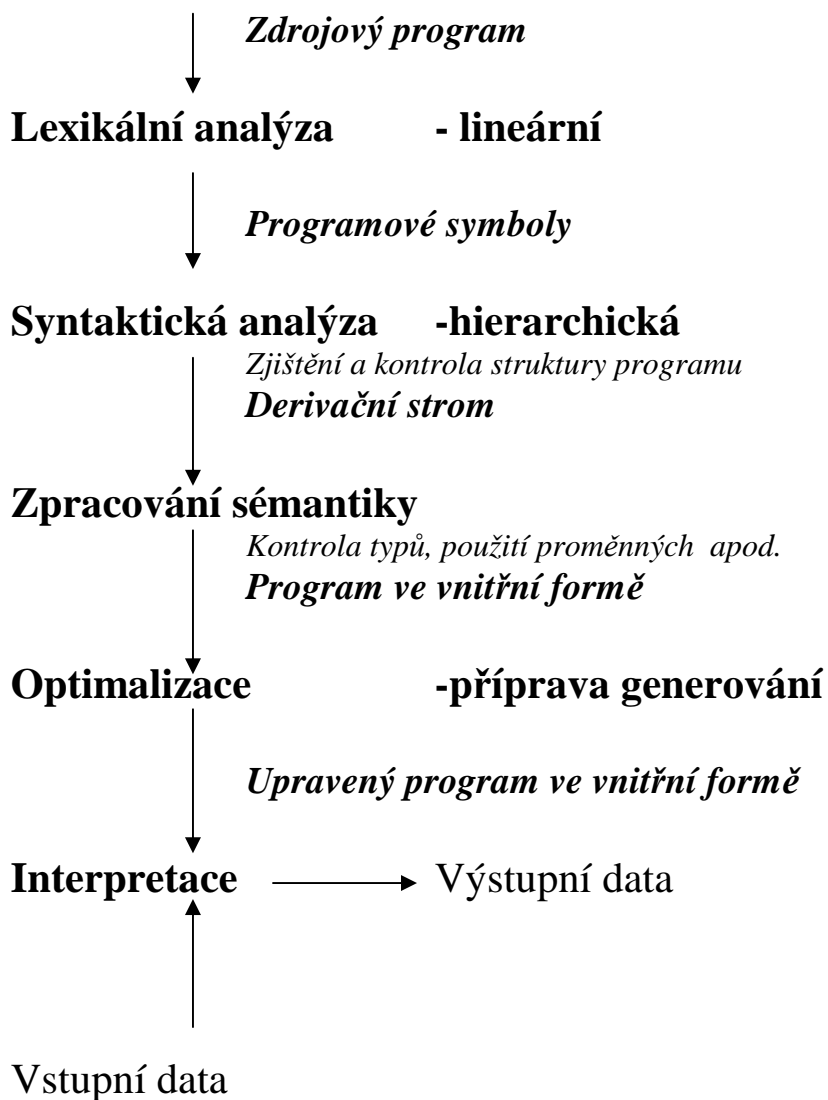
### Kompilátor:



Všechny části spolupracují s pracovními tabulkami překladače.  
Základní tabulkou kompilátoru i interpretu je  
Tabulka symbolů

Výhodou kompilátoru je rychlá exekuce programu

## Interpret



**Výhodou interpretu je:**

**eliminace kroků cyklu (Editace → překlad → sestavení → exekuce )**

**Snazší realizace ladících mechanismů (zachování původních jmen symbolů)**

## **Vícefázový / 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 možnost zpracování chyb a ladění (pro výuku)

## **Co má vliv na strukturu překladače**

- 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**

- Formální specifikace jazyka ⇒ možnost automatického generování testů,
- Systematické testování ⇒ regresní testy = sada testů doplňovaná o testy na odhalené chyby. Po každé změně v překladači se provedou všechny testy a porovnají se výstupy s předešlými.

## Vnitřní jazyky překladače

- **Postfixová notace (operátory bezprostředně následují za svými operandy, pořadí operandů je zachováno)**

Vyjádřuje precedenci operátorů, nepotřebuje závorky

$$\text{Př.1} \quad a + b \quad \rightarrow \quad a \ b \ +$$

$$\text{Př.2} \quad ( a + b ) * ( c + d ) \quad \rightarrow \quad a \ b \ + \ c \ d \ + \ *$$

Postfixový zápis nepotřebuje (a nemá) závorky

Postfix je elegantně vyhodnotitelný zásobníkem:

- 1) Čti symbol postfixového řetězce,
- 2) Je-li symbolem operand, ulož jej do zásobníku.
- 3) Je-li symbolem operátor, proved' jeho operaci nad vrcholem zásobníku a výsledek vlož do zásobníku
- 4) Jdi na 1).

Po přečtení a vyhodnocení celého řetězce je výsledek uložen v zásobníku (princip interpretace).

Pro př.2

čte a	čte b	čte +	čte c	čte d	čte +	čte *
a	b	c	d	c	c+d	(a+b)*(c+d)
a	a	a+b	a+b	a+b	a+b	(a+b)*(c+d)

**Pozn.** Musíme umět vyjádřit i jiné než konstrukce pro výrazy.

- **Prefixová notace (operátory bezprostředně předchází operandy, pořadí operandů je zachováno)**

Vyjádřuje precedenci operátorů, nepotřebuje závorky

$$a + b \quad \rightarrow \quad + \ a \ b$$

$$( a + b ) * ( c + d ) \quad \rightarrow \quad * \ + \ a \ b \ + \ c \ d$$

Prefixový zápis nemá závorky

**!Pozor, není to vždy zrcadlový obraz operátorů z postfixu**

**!!! pořadí operandů u postfixu i prefixu zůstává zachováno, mění se pořadí operátorů!!!**

Zkusme na tabuli př.  $A = - B * C + D$



- Víceadresové instrukce (čtveřice / trojice)

**Čtveřice** operátor, operand, operand, výsledek  
 Např. +, a, b, Výsledek  
 Potřeba přidělovat paměť pomocným prom.

---

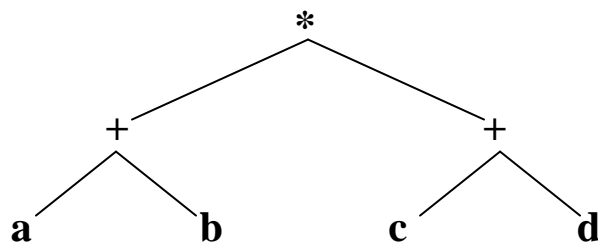
Př 2) ( a + b ) \* ( c + d )

tvar	význam
+ , a , b , Pom1	Pom1 = a + b
+ , c , d , Pom2	Pom2 = c + d
* , Pom1 , Pom2 , Vysl	Vysl = Pom1 * Pom2

**Trojice** odkládají potřebu přidělovat paměť pomocným proměnným v době generování víceadresových instrukcí. Vztahují výsledek operace k číslu trojice

Př 2)            1) +, a, b  
                   2) +, c, d  
                   3) \*, (1), (2)

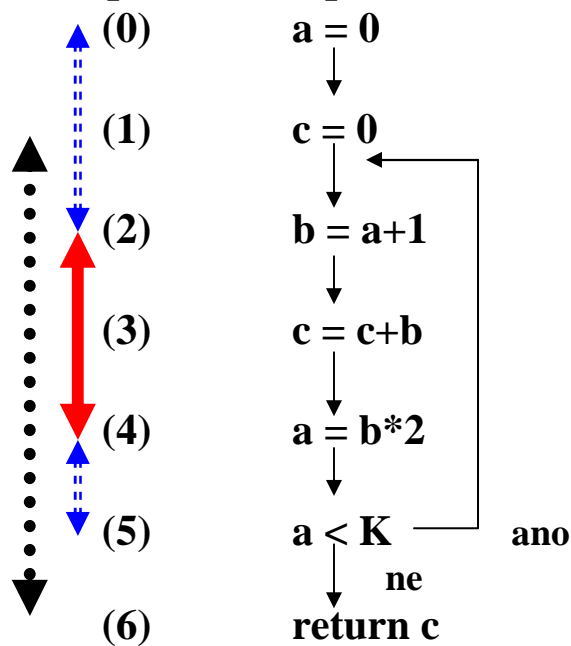
Vyjadřují abstraktní syntaktický strom



## Optimalizace

- Optimalizace cyklů
- Redukce počtu registrů
- ...a další

Př. Optimalizace paměti.

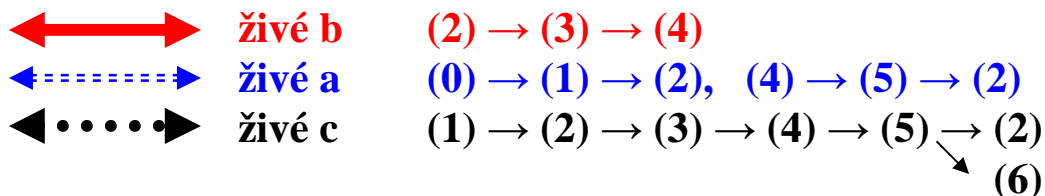


program

```

a = 0; c = 0;
L : b = a + 1;
  c = c + b;
  a = b * 2;
  if a < K goto L;
  return c;

```



?Kolik potřebujeme registrů pro proměnné a, b, c, když K je konstanta?

- zjištění živých a neživých proměnných v data flow diagramu,
- vytvoření interferenčního grafu,
- barvení grafu.

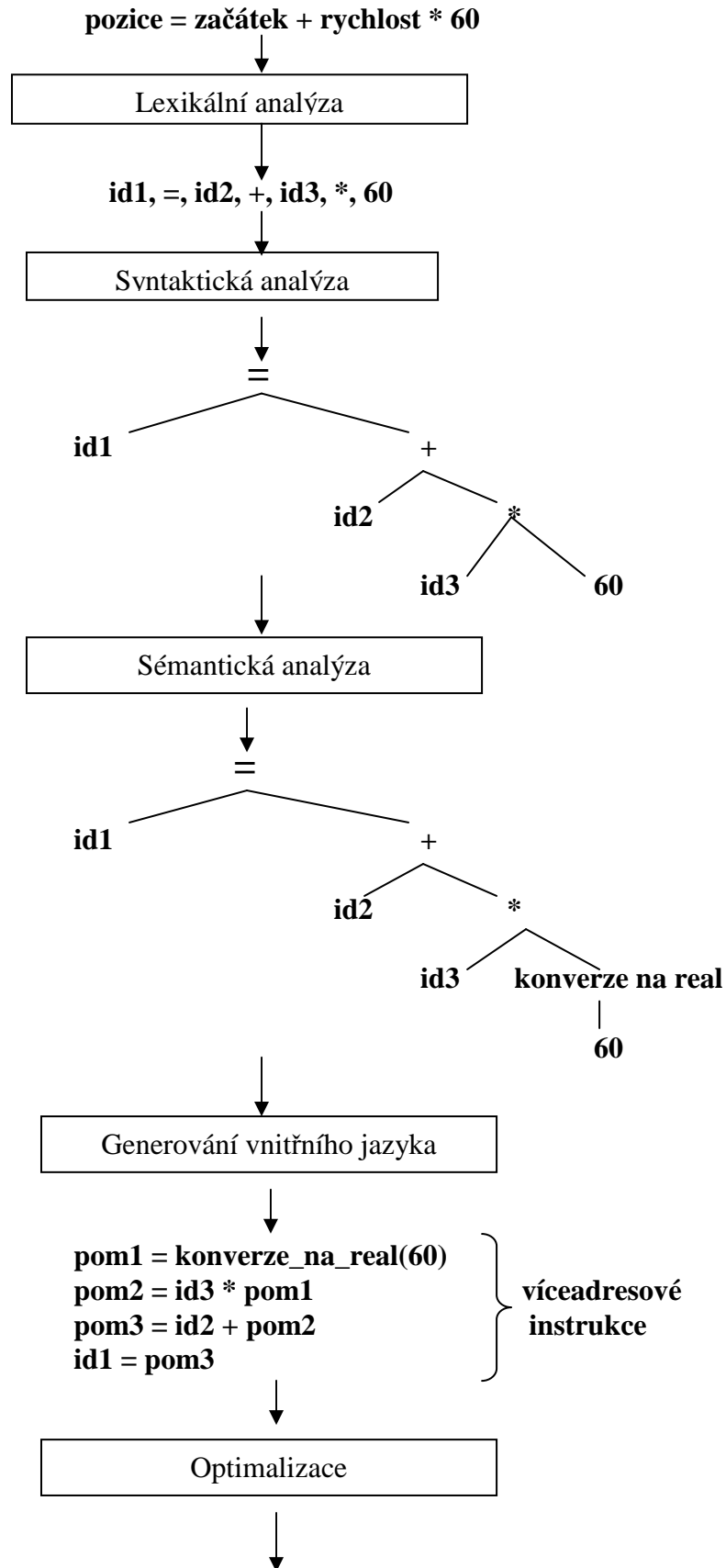
počet potřebných barev = počet potřebných registrů

	a	b	c
a			x
b			x
c	x	x	

Matrice interferencí

Výsledek 2

## Př. překladu příkazu



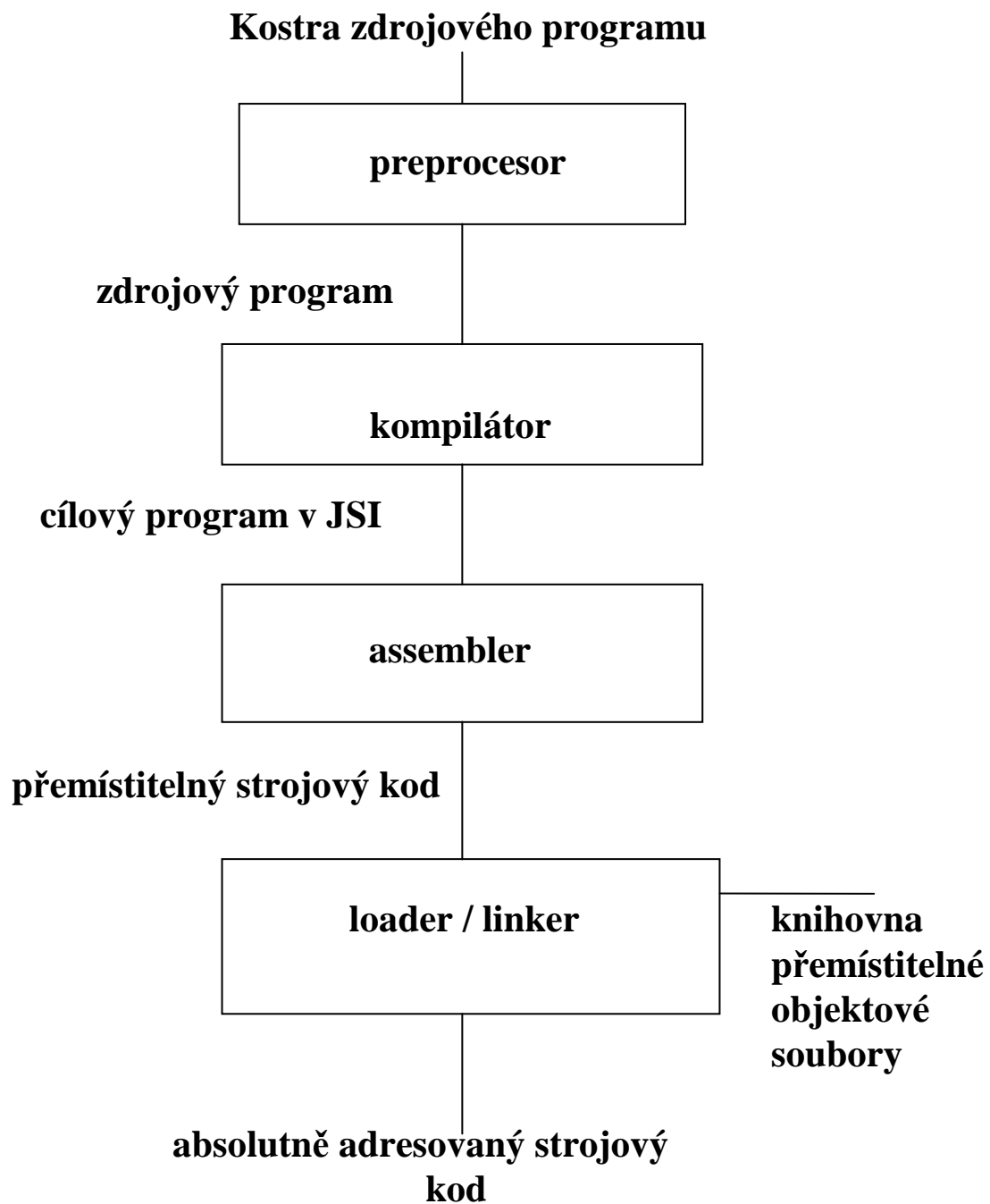
↓  
**pom1 = id3 \* 60.0**  
**id1 = id2 + pom1**

↓  
Generování kódu

↓  
**MOVF id3, R2**  
**MULF #60.0, R2**  
**MOVF id2, R1**  
**ADDF R2, R1**  
**MOVF R1, id1**

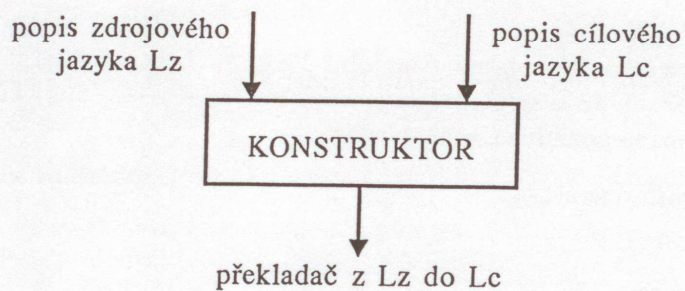
### Tabulka symbolů

<b>1</b>	<b>pozice</b>	<b>...</b>
<b>2</b>	<b>zacatek</b>	<b>...</b>
<b>3</b>	<b>rychlost</b>	<b>...</b>
<b>4</b>		
<b>5</b>		

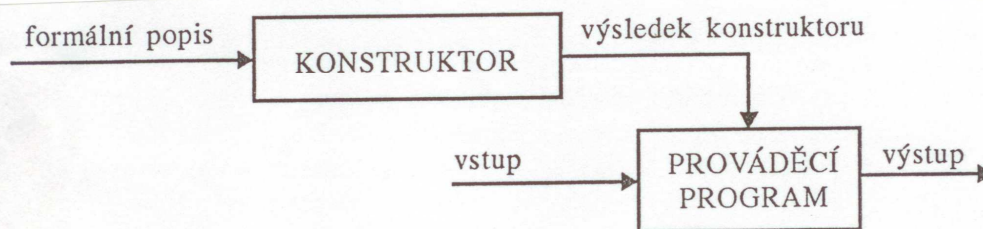


**Obr. Systém zpracování jazyka**

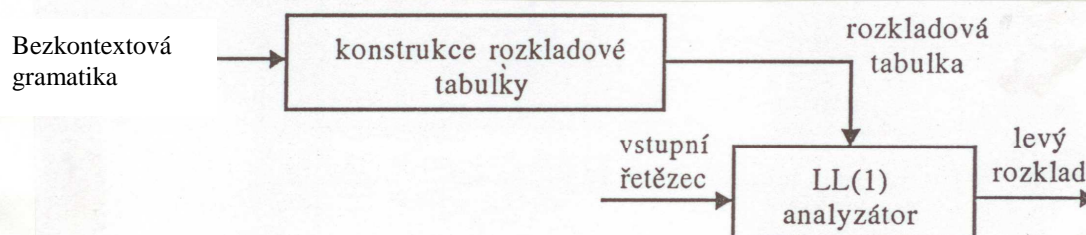
# Principiální možnost automatizace konstrukce překladače



Obr. 1.12: Struktura dokonalého systému pro konstrukci překladače

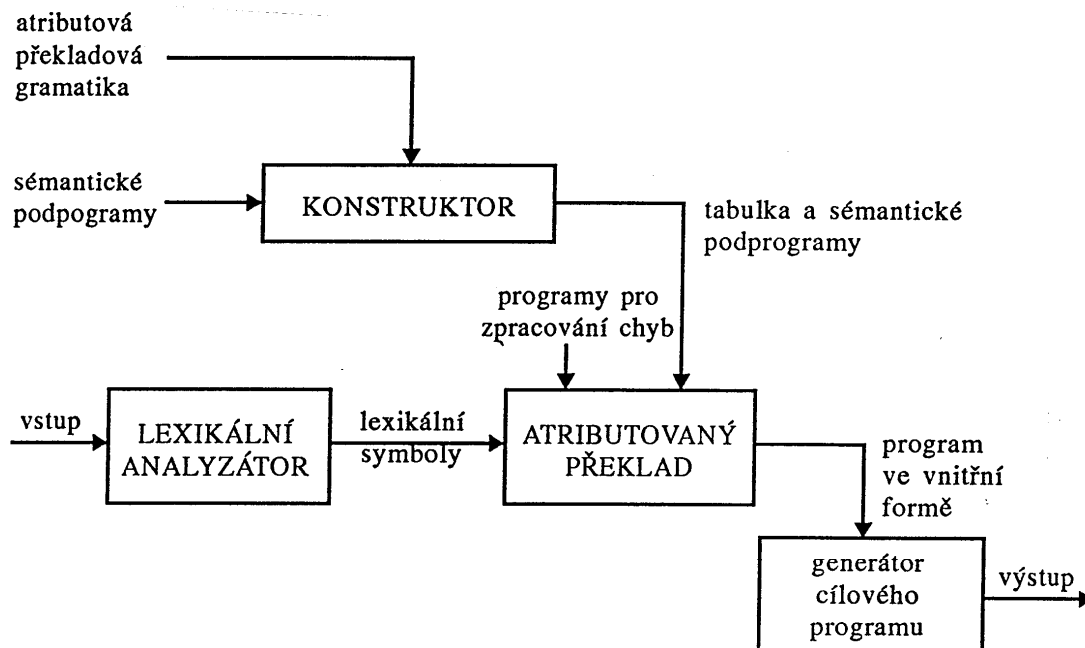


Obr. 1.13: Dvojice konstruktor—prováděcí program

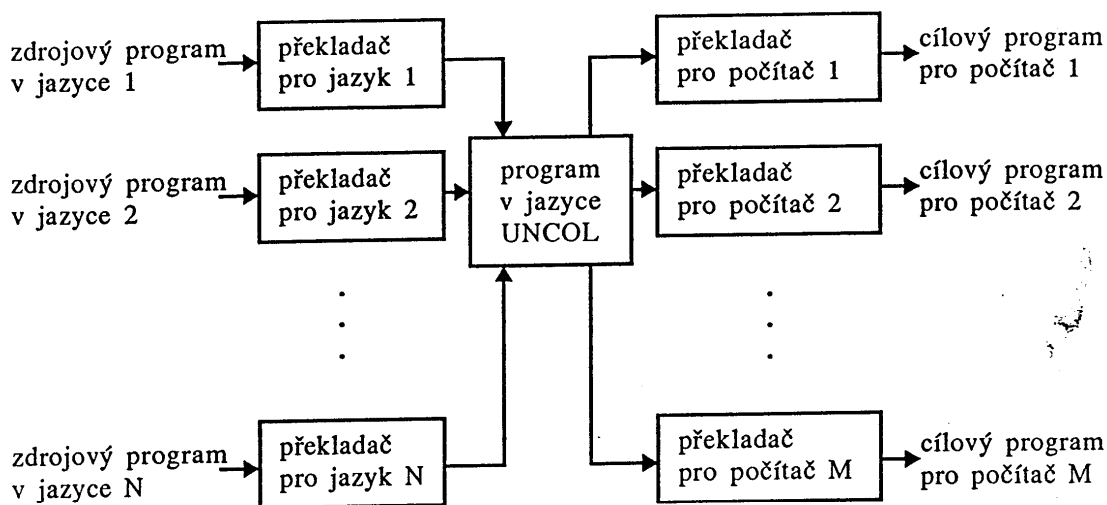


Obr. 1.14: Systém pro konstrukci LL(1) analyzátoru

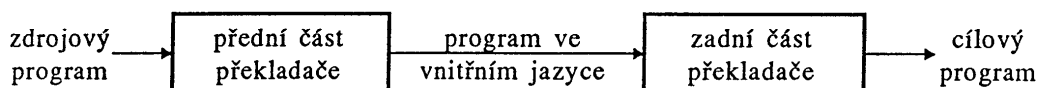
## Jaké prostředky k tomu máme a co bychom chtěli



Obr. Systém automatické konstrukce syntaktického analyzátoru s připojeným zpracováním sémantiky



Obr. Základní myšlenka univerzálního vnitřního jazyka UNCOL



Obr. Přední a zadní část překladače

<http://osteele.com/tools/rework/#>  
<http://dinosaur.compilertools.net/>

**Javascript, Python, PHP, Ruby**  
**Lex, Flex, Yacc, Bison stránky**

# The Lex & Yacc Page

[Overview](#) | [Lex](#) | [Yacc](#) | [Flex](#) | [Bison](#) | [Tools](#) | [Books](#)

## OVERVIEW

A compiler or interpreter for a programming language is often decomposed into two parts:

1. Read the source program and discover its structure.
2. Process this structure, e.g. to generate the target program.

*Lex* and *Yacc* can generate program fragments that solve the first task.

The task of discovering the source structure again is decomposed into subtasks:

1. Split the source file into tokens (*Lex*).
  2. Find the hierarchical structure of the program (*Yacc*).
- [A First Example: A Simple Interpreter](#)

## LEX

### Lex - A Lexical Analyzer Generator

*M. E. Lesk and E. Schmidt*

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

- [Online Manual](#)
- [PostScript](#)
- [Lex Manual Page](#)

## YACC

### Yacc: Yet Another Compiler-Compiler



*Stephen C. Johnson*

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

- [Online Manual](#)
- [PostScript](#)
- [Yacc Manual Page](#)

## FLEX

### **Flex, A fast scanner generator**

*Vern Paxson*

flex is a tool for generating scanners: programs which recognized lexical patterns in text. flex reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. flex generates as output a C source file,

- [Online Manual](#)
- [PostScript](#)
- [Flex Manual Page](#)
- [Download Flex from ftp://prep.ai.mit.edu/pub/gnu/](ftp://prep.ai.mit.edu/pub/gnu/)

## BISON

### **Bison, The YACC-compatible Parser Generator**

*Charles Donnelly and Richard Stallman*

Bison is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar. Once you are proficient with Bison, you may use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.

- [Online Manual](#)
- [PostScript](#)
- [Bison Manual Page](#)
- [Download Bison from ftp://prep.ai.mit.edu/pub/gnu/](ftp://prep.ai.mit.edu/pub/gnu/)

## TOOLS

Other tools for compiler writers:

- [Compiler Construction Kits](http://catalog.compilertools.net/kits.html) <http://catalog.compilertools.net/kits.html>
- [Lexer and Parser Generators](http://catalog.compilertools.net/lexparse.html) <http://catalog.compilertools.net/lexparse.html>
- [Attribute Grammar Systems](http://catalog.compilertools.net/attribute.html) <http://catalog.compilertools.net/attribute.html>
- [Transformation Tools](http://catalog.compilertools.net/trafo.html) <http://catalog.compilertools.net/trafo.html>
- [Backend Generators](http://catalog.compilertools.net/backend.html) <http://catalog.compilertools.net/backend.html>
- [Program Analysis and Optimisation](http://catalog.compilertools.net/optim.html) <http://catalog.compilertools.net/optim.html>
- [Environment Generators](http://catalog.compilertools.net/env.html) <http://catalog.compilertools.net/env.html>
- [Infrastructure, Components, Tools](http://catalog.compilertools.net/infra.html) <http://catalog.compilertools.net/infra.html>
- [Compiler Construction with Java](http://catalog.compilertools.net/java.html) <http://catalog.compilertools.net/java.html>

## BOOKS



### [Lex & Yacc](#)

John R. Levine, Tony Mason, Doug Brown  
Paperback - 366 pages 2nd/updated edition  
O'Reilly & Associates  
ISBN: 1565920007



### [Compilers: Principles, Techniques, and Tools](#)

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman  
Addison-Wesley Pub Co  
ISBN: 0201100886



### [Modern Compiler Implementation in C](#)

Andrew W. Appel, Maia Ginsburg  
Hardcover - 560 pages Rev expand edition  
Cambridge University Press  
ISBN: 052158390X