

# Programovací jazyky a překladače

## Úvod do překladačů

Ing. Marek Běhálek  
Katedra informatiky FEI VŠB-TUO  
A-1018 / 597 324 251

<http://www.cs.vsb.cz/behalek>  
[marek.behalek@vsb.cz](mailto:marek.behalek@vsb.cz)

Materiály vycházejí z původních prezentací  
doc. Ing. Miroslava Beneše Ph.D.



## Osnova přednášek

- Úvod - Obecné vlastnosti překladačů
- Lexikální analýza
- Syntaktická analýza
- LL(1) gramatiky
- Syntaxí řízený překlad
- JavaCC
- Tabulka symbolů
- Struktura programu v době běhu
- Vnitřní reprezentace
- Optimalizace
- Generování cílového kódu



## Úvod - Úloha překladače (1)



- **Překlad jednoho jazyka na druhý**

- Co je to jazyk?
  - Přírozený jazyk – složitá, nejednoznačná pravidla
  - Formální jazyk - popsán *gramatikou*
- Co je to překlad?
  - Zobrazení  $T : L1 \rightarrow L2$
  - L1: zdrojový jazyk (např. C++)
  - L2: cílový jazyk (např. strojový kód P4)

## Úvod - Úloha překladače (2)



- **Vyhledání chyb ve zdrojovém textu**

- Snažíme se co nejvíce chyb objevit v době překladu
- Zotavení po chybě – překladač je schopen pokračovat v činnosti a najít další chyby
- Oprava chyb – náročné, nepoužívá se

- **Vytvoření informací pro ladění programu**

- Jména proměnných a funkcí
- Odkazy do zdrojového textu

## Úvod - Zdrojový jazyk



- **Přirozený jazyk**
  - Předmět zájmu (počítačové) lingvistiky
- **Programovací jazyk** ←
  - C, C++, Java, C#, Prolog, Haskell
- **Speciální jazyk**
  - Jazyky pro popis VLSI prvků (VHDL)
  - Jazyky pro popis dokumentů (LaTeX, HTML, XML, RTF)
  - Jazyky pro popis grafických objektů (PostScript)

Programovací jazyky a překladače

5

## Úvod - Cílový jazyk



- **Strojový jazyk**
  - Absolutní binární kód
  - Přemístitelný binární kód (.obj, .o)
  - Jazyk symbolických instrukcí
  - Vyšší programovací jazyk (např. C)
- **Jazyk virtuálního procesoru**
  - Java Virtual Machine
  - MSIL pro .NET

Programovací jazyky a překladače

6

# Úvod - Využití technologie překladačů



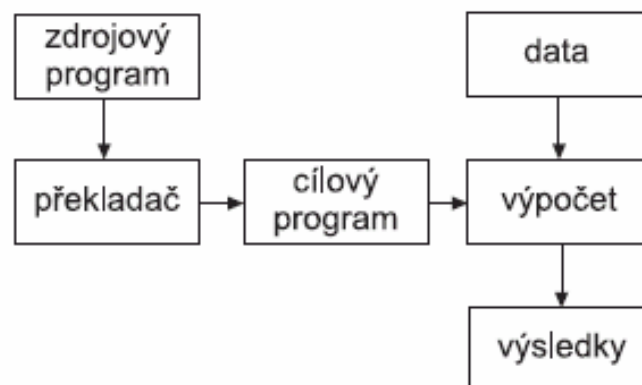
## Klasické překladače

- základní součást vývojových nástrojů v operačním systému
- křížové překladače (*cross compilers*) - vývoj programů pro vestavné systémy apod.

## Specializované překladače

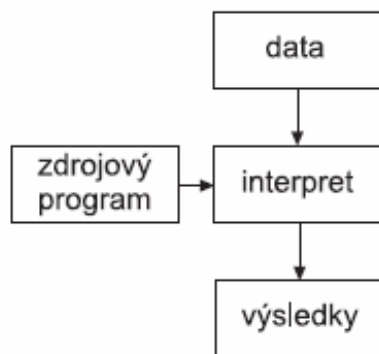
- systémy pro formátování textů (nroff, troff, LaTeX)
- silikonové překladače - popis a vytváření struktury VLSI obvodů
- příkazové interprety v operačním systému (shell)
- databázové systémy – dotazy, SQL
- reprezentace grafických objektů - jazyk PostScript
- skriptovací jazyky – možnost rozšíření určitého systému uživatelem

# Úvod - Typy překladačů (1) Kompilační překladač



## Úvod - Typy překladačů (2)

### Interpretační překladač



Programovací jazyky a překladače

9

## Úvod - Typy překladačů (3)

### Rozdíly mezi kompilátorem a interpretem



- Interpret je mnohem pomalejší než kompilátor
  - Je potřeba analyzovat zdrojový příkaz pokaždé, když na něj program narazí.
  - Interpret je 10 x až 100 x pomalejší.
- Interpret má ale i výhody.
  - Nezávislost na platformě.
  - Snadnější ladění – v době chyby máme k dispozici všechny informace.
  - Možnost měnit program za běhu - Smaltalk.

Programovací jazyky a překladače

10

## Úvod - Typy překladače (4)



- Inkrementální překlad
  - Umožňuje po drobné opravě přeložit jen změněnou část
  - Možnost provádění drobných změn během ladění programu
- Just-in-time překlad
  - Generování instrukcí virtuálního procesoru (Java VM - .class, .NET CLR – jazyk IL)
  - Překlad až v okamžiku volání podprogramu
  - Optimalizace podle konkrétního procesoru

## Úvod - Typy překladače (5)



- Zpětný překladač
  - Umožňuje získat zdrojový program z cílového (např. z .exe, .class)
    - disassembler (např. ILDASM v prostředí .NET)
    - decompiler (např. DJ Java Decompiler)
  - V některých státech (USA) není dovoleno (u nás **ano** – viz § 66 autorského zákona)
  - *Obfuscation* („zmatení“, také „duševní pomatenost“ – viz <http://slovniky.seznam.cz/>)
  - Transformace cílového programu komplikující zpětný překlad

## Úvod - Historie



- Základní principy překladačů už jsou přes 30 let stále stejné a tvoří součást jádra informatiky
- I nejstarší programovací jazyky jsou stále živé – tradice, investice, vývoj
- Z historie se můžeme poučit – inspirace, ověřené principy i chyby

## Úvod - Modely zdrojového programu

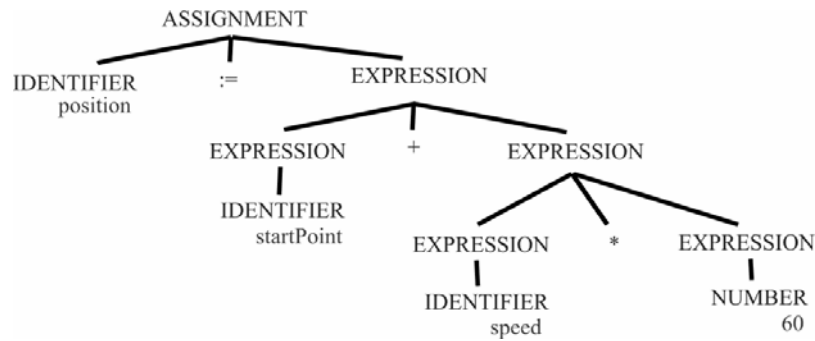


- **Vstup: Zdrojový program**
  - `position := startPoint + speed * 60;`
- **Lexikální analýza**
  - `<ID,position> <:=,> <ID,startPoint> <+,> <ID,speed> <*,> <INT,60>`

# Úvod - Modely zdrojového programu



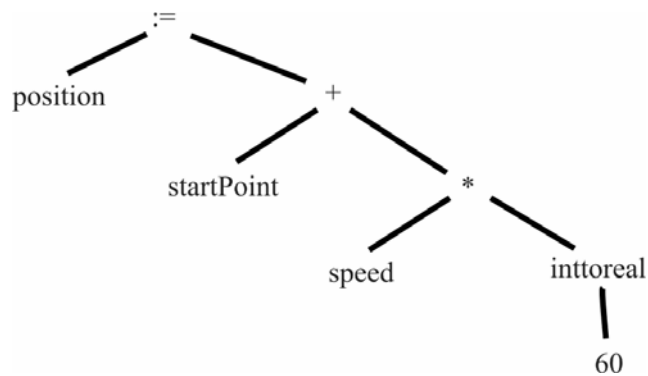
- **Syntaktická analýza**



# Úvod - Modely zdrojového programu



- **Kontextová analýza**





## Úvod - Modely zdrojového programu



- **Generování mezikódu**

```
temp1 := inttoreal(60)
temp2 := speed * temp1
temp3 := startPoint + temp2
position := temp3
```

- **Optimalizace**

```
temp1 := speed * 60.0
position := startPoint + temp1
```

## Úvod - Modely zdrojového programu



- **Generování cílového programu**

```
fld  qword ptr [_speed]
fmul dword ptr [00B2]      ; 60.0
fadd qword ptr [_startPoint]
fstp qword ptr [_position]
```

## Úvod - Organizace překladačů



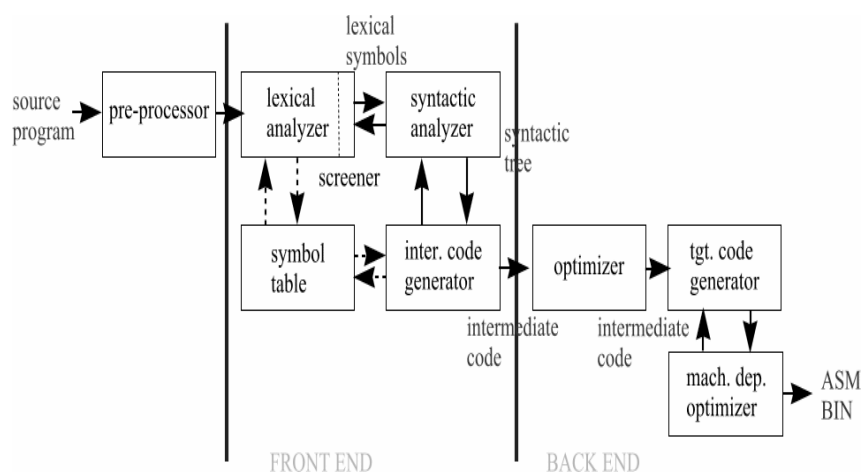
### • FÁZE

- Logická dekompozice (lexik., synt. analýza, ...)

### • PRŮCHOD

- Čtení vstupního souboru, zpracování, zápis výstupního souboru
- Může zahrnovat více fází
- Může být částí jedné fáze (např. optimalizace)

## Úvod - Struktura překladače



## Úvod - Vliv na strukturu překladače



- Vlastnosti zdrojového a cílového jazyka
- Paměť dostupná pro překlad
- Rychlost / velikost překladače
- Rychlost / velikost cílového programu
- Požadavky na ladění
- Detekce chyb
- Zotavení
- Velikost projektu – velikost týmu, termíny

## Úvod - Jednoprůchodový překlad



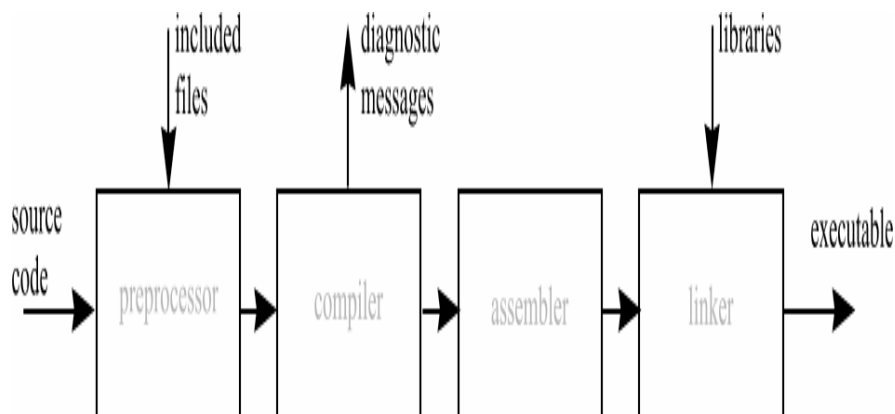
- Všechny fáze probíhají v rámci jediného čtení zdrojového textu programu
- Omezená možnost kontextových kontrol
- Omezená možnost optimalizace
- Lepší možnost zpracování chyb a ladění
- **Využití:**  
Výuka – rychlý překlad (program častěji překládáme než spouštíme)

## Úvod - Optimalizující překladače



- Více průchodů
- Často několik optimalizačních průchodů
- Možnost ovlivnění rozsahu prováděných optimalizací programátorem
- **Paralelizující překladače**
  - rozklad na úseky s možností paralelního provádění

## Úvod - Další pomocné programy



## Úvod - Další pomocné programy



- Editory
- Ladění - td, codeview, sdb, gdb
- Analýza obrazu paměti (post mortem dump)
- Reverzní překladače
- Automatické formátování textu
- Prohlížeče programu
- Profilování programu
- Správa verzí (CVS)
- Správa projektů (make, ant)
- Testování (junit)
- Správa knihoven
- Integrovaná vývojová prostředí - IDE (Delphi, MS Visual Studio, Eclipse, NetBeans, KDeveloper, ...)

## Úvod - Testování a údržba překladače



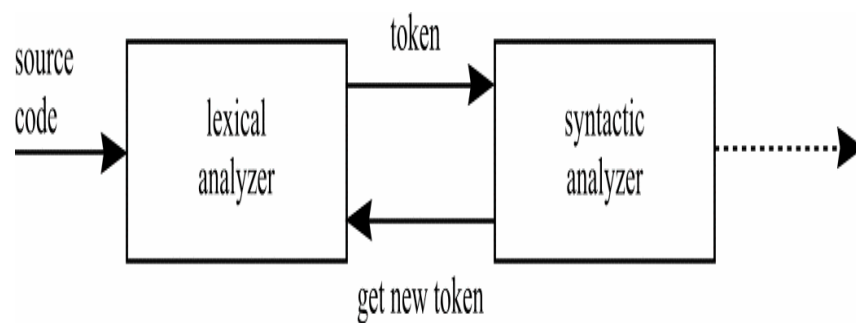
- Překladač **musí** vytvářet *korektní* (?) výstup
- **Formální specifikace** (? Ada, M-2 ISO)
  - -> Generátory testů
  - -> Automatické testování
- **Systematické testování**
- **Testování zátěže**
- **Údržba**: hlavně změny ve specifikaci jazyka styl programování, dokumentace

## Úvod - Implementace překladače



- **Retargetting – přenos na jinou cílovou platformu**
  - Strojově nezávislý mezikód
  - Interpretační překladače
- **Přizpůsobení existujícího překladače**
  - Podobné jazyky
- **Využití generátorů překladačů**
- **Zřídka se implementuje zcela od začátku**

## Lexikální analýza - Rozhraní lexikálního analyzátoru



## Lexikální analýza - Úkoly



- Čtení zdrojového textu
- Sestavování symbolů
- Odstranění mezer a poznámek
- Normalizace symbolů (velká/malá písmena, spec. znaky, ...)
- Interpretace direktiv překladače
- Uchování informací pro hlášení chyb
- Zobrazení protokolu o překladu

## Lexikální analýza - Proč řešit lexikální analýzu samostatně?



- Jednodušší návrh překladače
  - Konvence pro mezery a poznámky
- Vyšší efektivita
  - Specializované algoritmy
- Lepší přenositelnost
  - Zvláštnosti vstupní abecedy

??( = [

## Lexikální analýza - Základní pojmy



- **Lexém** – slovo nad abecedou
- **Kategorie symbolů** – identifikátor, číslo, relační operátor, levá závorka, ...
- **Atributy symbolu** – řetězec, hodnota, kód operátoru, ...
- **Reprezentace symbolu** – dvojice (*kategorie*, *atribut*)

## Lexikální analýza - Příklady lexikálních symbolů



<b>Symbol</b>	<b>Lexém</b>	<b>Vzor</b>
<b>const</b>	const	const
<b>relation</b>	<, <=, ..., >=	<>
<b>id</b>	{ltr}({ltr} {dig})*	pi, D2
<b>num</b>	{dig}+	0, 123



## Lexikální analýza - Reprezentace symbolů



```
// kategorie symbolů
enum Symbol { IdSym, NumSym, RelOpSym, DotSym, ... };

// kategorie operátorů
enum Oper { LthOp, GthOp, LeqOp, ... };

// atributy symbolů
union LexAttr {
    char* id;
    int num;
    Oper relop;
    ...
};

// reprezentace lexikálního symbolu
struct Token {
    Symbol sym; // kategorie
    LexAttr attr; // atribut
};
```

Programovací jazyky a překladače

33

## Lexikální analýza - Specifikace symbolů (1)



- **Popis běžným jazykem**
  - Identifikátor je posloupnost písmen a číslic začínající písmenem'
- **Regulární (lineární) gramatika**
  - $I \rightarrow pX$
  - $X \rightarrow pX \mid cX \mid p \mid c$
- **Regulární výrazy a definice**
  - $p(p \mid c)^*$
  - Nejjednodušší a nejčastěji používané

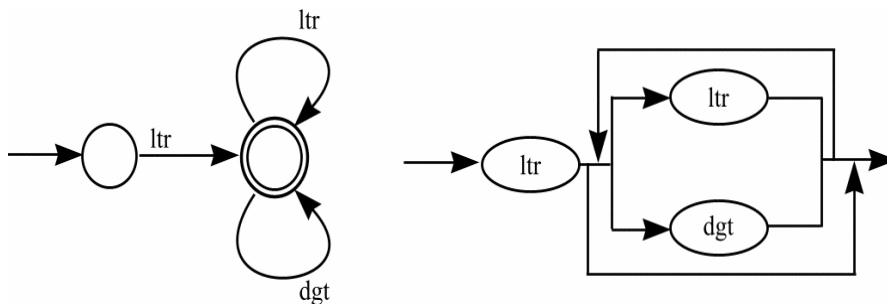
Programovací jazyky a překladače

34

## Lexikální analýza - Specifikace symbolů (2)



- Graf přechodů konečného automatu
  - (syntaktický graf)



Programovací jazyky a překladače

35

## Lexikální analýza - Specifikace symbolů (3)



- Lexikální symboly lze obvykle popsat *regulárními jazyky*
- **Co nedokážeme popsat?**
  - Zanořené konstrukce (závorky)
  - Opakované konstrukce  $\{wcw | w \in \{a,b\}^*\}$
  - Zadaný počet opakování
    - $nHa_1a_2\dots a_n$  (FORTRAN)

Programovací jazyky a překladače

36

## Lexikální analýza - Příklad regulárních výrazů



- Přesná definice regulárních výrazů - UTI
- Následující příklady demonstrují běžné konvence používané v různých nástrojích
  - $[A-Z]([A-Z]|[0-9])^* \equiv [A-Z][A-Z0-9]^*$
  - $[0-9]+\.[0-9]+(E[-+]?[0-9]+)?$   
|  $[0-9]+E[-+]?[0-9]^+$

## Lexikální analýza - Regulární definice (1)

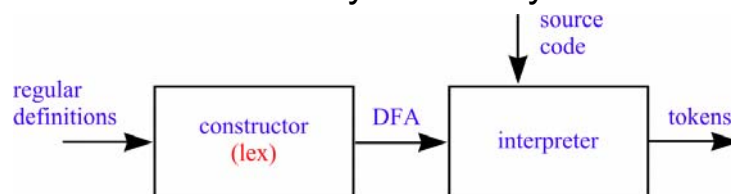


- Pojmenované regulární výrazy
  - $d_1 \rightarrow r_1$
  - $d_2 \rightarrow r_2$
  - ...
  - $d_n \rightarrow r_n$
- různá jména reg. výrazy nad  $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$

## Lexikální analýza - Regulární definice (2)



- Pojmenované regulární výrazy
  - letter -> [A-Za-z]
  - digit -> [0-9]
  - id -> letter (letter | digit)\*
- Použití: konstruktory lex. analyzátorů



Programovací jazyky a překladače

39

## Lexikální analýza - Konečné automaty (1)

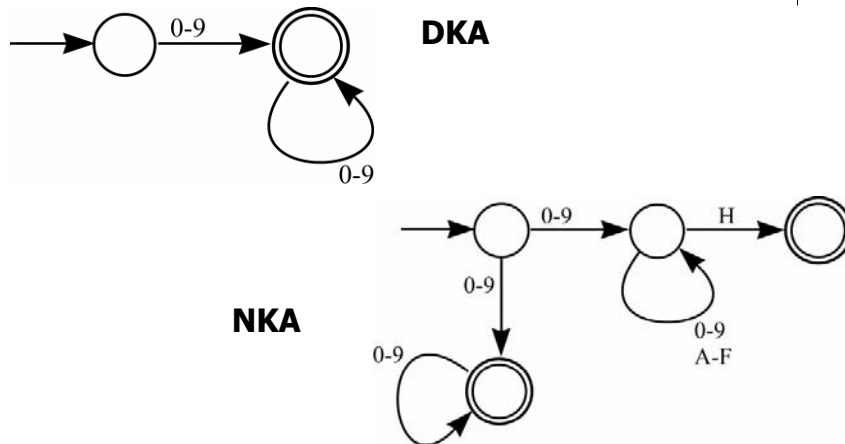


- $(Q, \Sigma, f, q_0, F)$ 
  - $Q$  – konečná množina stavů
  - $\Sigma$  - vstupní abeceda
  - $f$  – přechodová funkce
  - $q_0$  – počáteční stav
  - $F$  – množina koncových stavů
- $f: Q \times (\Sigma \cup \{e\}) \rightarrow 2^Q$       rozšířený NKA
- $f: Q \times \Sigma \rightarrow Q$                       DKA

Programovací jazyky a překladače

40

## Lexikální analýza - Konečné automaty (2)



Programovací jazyky a překladače

41

## Lexikální analýza - Algoritmy pro transformaci



- Regulární gramatika  $\leftrightarrow$  konečný automat
  - korespondence pravidel gramatiky a přechodové funkce
- Regulární výraz  $\rightarrow$  konečný automat
  - skládání primitivních KA, převod na DKA, minimalizace
  - stromová reprezentace
  - důležité pro konstruktory lexikálních analyzátorů
- Konečný automat  $\rightarrow$  regulární výraz
  - soustava algebraických rovnic
$$X = aX + b \Rightarrow X = a^*b$$
  - derivace regulárních výrazů

Programovací jazyky a překladače

42

## Lexikální analýza - Konečný automat pro lexikální analýzu



- Zpracování začíná vždy prvním dosud nezpracovaným znakem ze vstupu
- Zpracování končí, je-li automat v koncovém stavu a pro další vstupní znak již neexistuje žádný přechod (maximal match):
  - 123 není 12 následované 3
- Není-li v nekoncovém stavu přechod možný, vrací se automat do posledního dosaženého koncového stavu nebo je chyba:
  - < <= <<      1. 1.2 1..5

## Lexikální analýza - Speciální případy (1)



- **akce po přijetí symbolu**
  - samostatný koncový stav pro každou kategorii
  - výpočet hodnot atributů z lexému
- **klíčová slova**
  - koncový stav pro každé klíčové slovo – mnoho stavů
  - obvykle jako id, pak následuje rozlišení tabulkou klíč. slov

## Lexikální analýza - Speciální případy (2)



- **komentáře**

- uzavřená cesta procházející poč. stavem
- diagnostika – neukončený komentář
- dokumentační komentář – Javadoc
- zanořené komentáře

- **znakové řetězce**

- escape sekvence ‘\n’
- ukončení řádku v řetězci je obvykle chyba
- Unicode

## Lexikální analýza - Implementace lexikálního analyzátoru



- **Přímá**

- Efektivita na úkor složitosti návrhu
- Stav je reprezentován pozicí v programu

- **Simulace konečného automatu**

- Vhodné spíše pro konstruktory

- **Využití konstruktoru**

- Snadná modifikovatelnost
- Především v počátečních fázích implementace
- LEX (FLEX), JavaCC

## Lexikální analýza - Přímá implementace



```
for(;;) {
    skipSpaces();
    skipNote();
    if( isEof ) return Tokens.EOF;
    if( Character.isLetter(ch) ) {
        StringBuffer buf = new StringBuffer();
        do { buf.append(ch); getch();
        } while( !isEof && Character.isLetterOrDigit(ch) );
        stringAttr = buf.toString();
        if( stringAttr.compareToIgnoreCase("div") == 0 ) return Tokens.DIV;
        if( stringAttr.compareToIgnoreCase("mod") == 0 ) return Tokens.MOD;
        return Tokens.IDENT;
    }
    if( Character.isDigit(ch) ) {
        StringBuffer buf = new StringBuffer();
        do { buf.append(ch); getch();
        } while( !isEof && Character.isDigit(ch) );
        numberAttr = Integer.parseInt(buf.toString());
        return Tokens.NUMBER;
    }
}
```

Programovací jazyky a překladače

47

## Lexikální analýza - Implementace konečného automatu



- Tabulka + Přejchodová funkce + Interpret
  - výhodné jako výstup konstrukturu
- Přímý přepis do programu

Programovací jazyky a překladače

48



## Lexikální analýza – Využití konstruktoru



- Vstupem je obvykle sada regulárních definic

SKIP :

```
{  
  " " | "\r" | "\t"  
}
```

TOKEN :

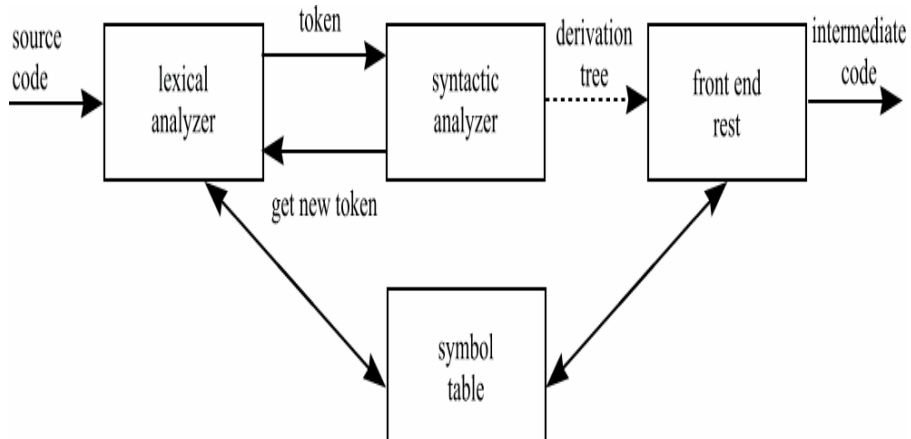
```
{  
  < ADD: "+" > | < SUB: "-" > | < MUL: "*" > | < DIV: "/" > | < MOD: "mod" >  
}
```

TOKEN :

```
{  
  < CONSTANT: ( <DIGIT> )+ > | < #DIGIT: ["0" - "9"] >  
}
```

- Konstruktor pak tento vstup zpracuje a vygeneruje implementaci lexikálního analyzátoru (obvykle využívá automaty)
  - Více informací v přednášce věnované JavaCC

## Syntaktická analýza - Rozhraní syntaktického analyzátoru



# Syntaktická analýza - Úkoly syntaktické analýzy



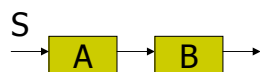
- **Rozpoznat syntaktické konstrukce**

- posloupnost  $AB$
- alternativa  $A \mid B$
- opakování  $AAA \dots$   $A, A, A, \dots$
- hierarchie  $E \rightarrow (E)$

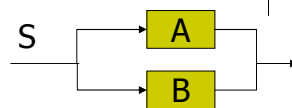
- **Vybudovat derivační strom**

- vnitřní uzly – neterminální symboly
- listy – terminální symboly

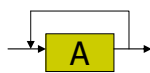
# Syntaktická analýza - Základní syntaktické konstrukce



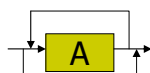
$$S \rightarrow AB$$



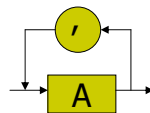
$$S \rightarrow A \mid B$$



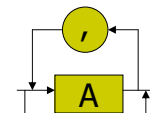
$$\begin{aligned} S &\rightarrow SA \mid A \\ S &\rightarrow AS \mid A \\ S &\rightarrow AS' \\ S' &\rightarrow AS' \mid e \end{aligned}$$



$$\begin{aligned} S &\rightarrow SA \mid e \\ S &\rightarrow AS \mid e \\ S &\rightarrow AS' \mid e \\ S' &\rightarrow AS' \mid e \end{aligned}$$

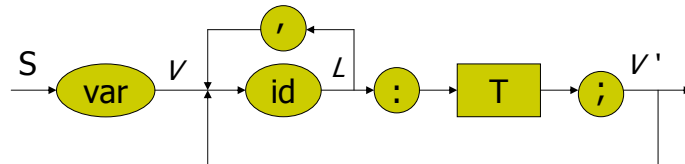


$$\begin{aligned} S &\rightarrow S, A \mid A \\ S &\rightarrow A, S \mid A \\ S &\rightarrow AS' \\ S' &\rightarrow ,AS' \mid e \end{aligned}$$



$$\begin{aligned} S &\rightarrow AS' \mid e \\ S' &\rightarrow ,AS' \mid e \end{aligned}$$

## Syntaktická analýza – Příklad (syntaktický graf)


$$S \rightarrow \text{var } V V' \quad V \rightarrow \text{id } L : T ;$$
$$V' \rightarrow V V' \quad L \rightarrow , \text{id } L$$
$$| e \quad | e$$

Programovací jazyky a překladače

53

## Syntaktická analýza - Metody syntaktické analýzy



- **Univerzální metody**
  - např. analýza s návraty
  - nejsou efektivní
- **Překlad shora dolů (LL gramatiky)**
  - vhodný pro „ruční“ implementaci
- **Překlad zdola nahoru (LR gramatiky)**
  - efektivní metoda pro všechny deterministicky analyzovatelné jazyky
  - využití zejména v konstruktorech překladačů

Programovací jazyky a překladače

54

## Syntaktická analýza - Bezkontextová gramatika



- **Bezkontextová gramatika je definovaná jako čtveřice  $G=(N, T, P, S)$** 
  - N – neterminální symboly
  - T – terminální symboly
  - P – pravidla ve tvaru -  $N \times (N \cup T)^*$
  - S – startovací neterminál

- **Příklad**

$G = (\{E, T, F\}, \{+, *, (, ), n\}, P, E)$

$P = \{$   
    E     ->   E + T   | T  
    T     ->   T \* F   | F  
    F     ->   (E )    | n  
    }

Programovací jazyky a překladače

55

## Syntaktická analýza - **Analýza shora dolů (1)**



- **Levá derivace**

$\underline{E} \Rightarrow \underline{E}+T \Rightarrow \underline{T}+T \Rightarrow \underline{F}+T \Rightarrow n+\underline{T} \Rightarrow$   
 $\Rightarrow n+\underline{T}*F \Rightarrow n+\underline{E}*F \Rightarrow n+n*\underline{E} \Rightarrow n+n*n$

- **Větná forma ...  $(N \cup T)^*$**

$n + n * F$

- **Věta ...  $T^*$**

$n + n * n$

Programovací jazyky a překladače

56

## Syntaktická analýza - Analýza shora dolů (2)



Vstup	Zásobník	Pravidlo
.n + n * n	.E	E -> E + T
.n + n * n	.E + T	E -> T
.n + n * n	.T + T	T -> F
.n + n * n	.F + T	F -> n
.n + n * n	.n + T	
. + n * n	. + T	P -> e
.n * n	.T	T -> T * F
.n * n	.T * F	T -> F
.n * n	.F * F	F -> n
.n * n	.n * F	
. * n	. * F	
.n	.F	F -> n
.n	.n	
.	.	

Programovací jazyky a překladače

57

## Syntaktická analýza - Analýza zdola nahoru (1)



- Pravá derivace

$$\begin{aligned}
 \underline{E} &\Rightarrow E + \underline{T} &\Rightarrow E + T * \underline{F} &\Rightarrow E + \underline{T} * n &\Rightarrow \\
 &\Rightarrow E + \underline{F} * n &\Rightarrow \underline{E} + n * n &\Rightarrow \underline{T} + n * n &\Rightarrow \\
 &\Rightarrow \underline{F} + n * n &\Rightarrow n + n * n &&
 \end{aligned}$$

Programovací jazyky a překladače

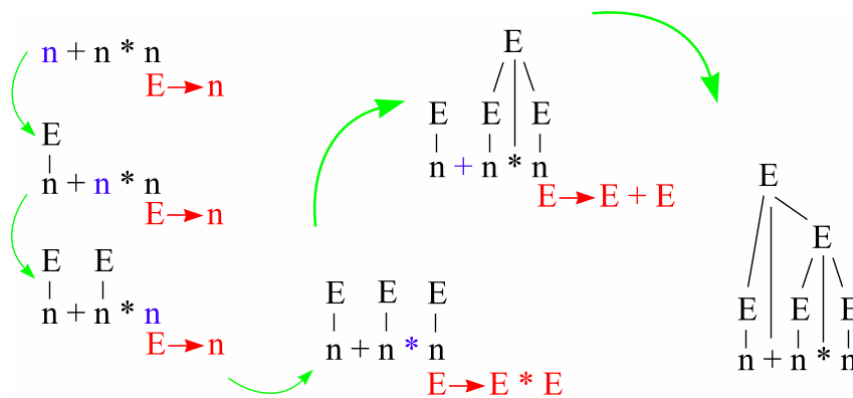
58

# Syntaktická analýza - Analýza zdola nahoru (2)



Zásobník	Vstup	Pravidlo
	.n + n * n	
n	.+ n * n	F -> n
F	.+ n * n	T -> F
E	.+ n * n	E -> T
E +	.n * n	
E + n	. * n	F -> n
E + F	. * n	T -> F
E + T	. * n	
E + T *	.n	
E + T * n	.	F -> n
E + T * F	.	T -> T * F
E + T	.	E -> E + T
E	.	

# Syntaktická analýza - Analýza zdola nahoru (3)



## Syntaktická analýza - LL(1) gramatiky



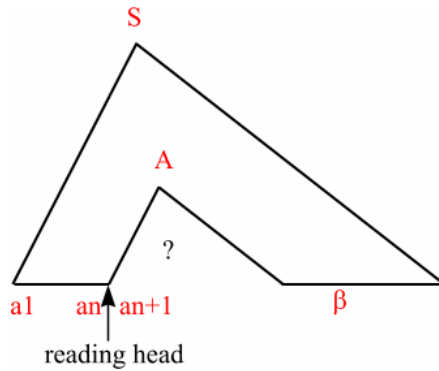
- LL(1)
  - L – Levý rozklad
  - L – Analýza zleva doprava
  - 1 – Rozhodování podle jednoho symbolu
- Deterministická analýza
- Analýza shora dolů
- Jednoduchá implementace

## Syntaktická analýza - LR(1) gramatiky



- Největší třída jazyků analyzovatelných deterministickými metodami
  - Každý jazyk generovaný LR(k) gramatikou lze generovat ekvivalentní LR(1) gramatikou
  - Neplatí pro LL(k)!
- Analýza se provádí *zdola nahoru* pomocí zásobníkového automatu

## LL(1) gramatiky - Motivace



- Vstup  
 $w = a_1 a_2 \dots a_n$
- Stav analýzy  
 $S \Rightarrow^* a_1 a_2 \dots a_j A B$
- $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$
- Kterou pravou stranu použít pro expanzi A?

Programovací jazyky a překladače

63

## LL(1) gramatiky - Příklad



- $A \rightarrow a B \mid b C \mid c$
- $A \rightarrow B a \mid c$   
 $B \rightarrow b B \mid d$
- $A \rightarrow a A \mid e$   
 $S \rightarrow b A c$
- Když  $a_{j+1} = a$ , pak použijeme  $A \rightarrow aB$ .
- Když  $a_{j+1} \in \{b, d\}$ , pak použijeme  $A \rightarrow Ba$ .  
 $FIRST(\alpha_i)$
- Když  $a_{j+1} \in \{c\}$ , pak použijeme  $A \rightarrow e$ .  
 $FOLLOW(A)$

Programovací jazyky a překladače

64





## LL(1) gramatiky - Definice

$G = (N, \Sigma, P, S) \quad \alpha \in (N \cup \Sigma)^*$

- **FIRST**( $\alpha$ ) ::=  $\{a \in \Sigma \mid \alpha \Rightarrow^* a\beta, \beta \in (\Sigma \cup N)^*\} \cup \{e \mid \alpha \Rightarrow^* e\}$
- **FOLLOW**(A) ::=  $\{a \in \Sigma \mid S \Rightarrow^* \alpha A \beta, a \in \text{FIRST}(\beta), \alpha, \beta \in (\Sigma \cup N)^*\}$



## LL(1) gramatiky - Příklad

$A \rightarrow BCb \mid aB$

$B \rightarrow bB \mid e$

$C \rightarrow cA \mid e$

$\text{FIRST}(BCb) = ?$

- $BCb \Rightarrow \mathbf{b}BCb$   
 $\Rightarrow Cb \Rightarrow \mathbf{c}Ab$   
 $\Rightarrow \mathbf{b}$
- $\text{FIRST}(BCb) = \{b, c\}$

$\text{FOLLOW}(B) = ?$

- $A \Rightarrow B|Cb$   
 $\Rightarrow bB|Cb$   
 $\Rightarrow \dots$   
 $\Rightarrow aB|$
- $\text{FIRST}(Cb) = \{b, c\}$
- $\text{FIRST}(e) = \{e\}$
- $\text{FOLLOW}(B) = \{b, c, e\}$

## LL(1) gramatiky - Definice



Množina symbolů generujících prázdné slovo

$$G = (N, \Sigma, P, S)$$

$$N_e ::= \{A \in N \mid A \Rightarrow^+ e\}$$

## LL(1) gramatiky - Algoritmus pro FIRST( $\alpha$ )



1.  $\alpha = a\beta, a \in \Sigma \quad \Rightarrow \text{FIRST}(\alpha) = \{a\}$
2.  $\alpha = e \quad \Rightarrow \text{FIRST}(\alpha) = \{e\}$
3.  $\alpha = A\beta, \quad \Rightarrow \text{FIRST}(\alpha) =$   
 $A \in N \setminus N_e \quad \cup_{i \in \{1..n\}} \text{FIRST}(\alpha_i)$   
 $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$
4.  $\alpha = A\beta, \quad \Rightarrow \text{FIRST}(\alpha) =$   
 $A \in N_e \quad (\cup_{i \in \{1..n\}} \text{FIRST}(\alpha_i) - \{e\})$   
 $\cup \text{FIRST}(\beta)$

## LL(1) gramatiky – Příklad (1)



$A \rightarrow BCb \mid aB$        $N_e = \{B, C\}$

$B \rightarrow bB \mid e$

$C \rightarrow cA \mid e$

- $\text{FIRST}(bB) = \{b\}$  (1)

- $\text{FIRST}(e) = \{e\}$  (2)

- $\text{FIRST}(cA) = \{c\}$  (1)

- $\text{FIRST}(BCb) = ((\text{FIRST}(bB) \cup \text{FIRST}(e)) \setminus \{e\}) \cup \text{FIRST}(Cb)$  (4)

## LL(1) gramatiky – Příklad (2)



- $\text{FIRST}(Cb) = ((\text{FIRST}(cA) \cup \text{FIRST}(e)) \setminus \{e\}) \cup \text{FIRST}(b)$  (4)

- $\text{FIRST}(b) = \{b\}$  (1)

- $\text{FIRST}(BCb) = \{b\} \cup \{b, c\} = \{b, c\}$

- Jiný možný algoritmus je založený na výpočtu tranzitivního uzávěru.

## LL(1) gramatiky - Algoritmus pro FOLLOW



1.  $e \in \text{FOLLOW}(S)$
2.  $A \rightarrow \alpha X \beta$   
 $\Rightarrow \text{FIRST}(\beta) \setminus \{e\} \subseteq \text{FOLLOW}(X)$
3.  $\beta \Rightarrow^* e$   
 $\Rightarrow \text{FOLLOW}(A) \subseteq \text{FOLLOW}(X)$

## LL(1) gramatiky - Příklad



$A \rightarrow BCb \mid aB$   
 $B \rightarrow bB \mid e$   
 $C \rightarrow cA \mid e$

- (1)  $e \in \text{FOLLOW}(A)$
- (2)  $\text{FIRST}(Cb) = \{b, c\} \subseteq \text{FOLLOW}(B)$   
 $\text{FIRST}(b) = \{b\} \subseteq \text{FOLLOW}(C)$
- (3)  $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$   
 $\text{FOLLOW}(B) \subseteq \text{FOLLOW}(B)$   
 $\text{FOLLOW}(C) \subseteq \text{FOLLOW}(A)$

- $\text{FOLLOW}(C) = \{b\}$
- $\text{FOLLOW}(A) = \{e, b\}$
- $\text{FOLLOW}(B) = \{e, b, c\}$

## LL(1) gramatiky – Omezení LL(1) gramatik



$A \rightarrow aA \mid aB$

$B \rightarrow b$

- Nejsme schopni na základě množiny FIRST určit, kterou pravou stranu neterminálu a použít.

$A \rightarrow Aa \mid e$

- Nevíme, kdy „ukončit“ rekurzi.

## LL(1) gramatiky – Definice LL(1) Gramatiky



- $G = (N, \Sigma, P, S)$
- Libovolné dvě levé derivace
  - $S \Rightarrow^* wA\beta \Rightarrow w\alpha_1\beta \Rightarrow^* wx$
  - $S \Rightarrow^* wA\beta \Rightarrow w\alpha_2\beta \Rightarrow^* wy$
- **FIRST(x) = FIRST(y)  $\Rightarrow \alpha_1 = \alpha_2$**

Pro danou větnou formu  $wA\beta$  a symbol  $a$  lze jednoznačně najít pravidlo  $A \rightarrow \alpha$  pro expanzi.

## LL(1) gramatiky – Důsledky



- $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$   
 $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$ , pro  $\forall i, j: i \neq j$   
*Podmínka FF*
- Pokud  $A \in N_e$  ( $\alpha_i \Rightarrow^* e$ ) pro nějaké  $i$ , pak  
 $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$ , pro  $\forall j: j \neq i$   
*Podmínka FFL*

## LL(1) gramatiky – Příklad



$A \rightarrow BCb \mid aB$  *není LL(1) gramatika*

$B \rightarrow bB \mid e$

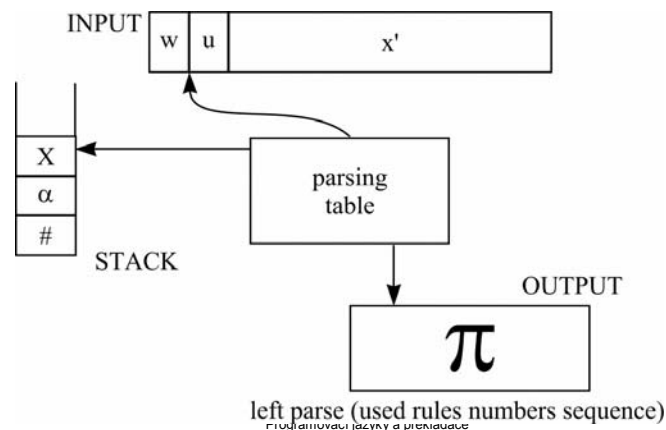
$C \rightarrow cA \mid e$

- $\{b, c\} \cap \{a\} = \emptyset$   
 $\Rightarrow$  Podmínka FF platí
- $(\text{FOLLOW}(B) = \{e, b, c\})$   
 $\cap (\text{FIRST}(bB) = \{b\}) = \{b\}$   
 $\Rightarrow$  Podmínka FFL neplatí

## LL(1) gramatiky – Syntaktická analýza LL(1) jazyků (1)



Prediktivní syntaktická analýza zás. automatem



77

## LL(1) gramatiky – Syntaktická analýza LL(1) jazyků (2)



- Konfigurace  $(x, X\alpha, \pi)$ 
  - $x$  – nezpracovaný vstup,  $x=ux'$ ,  $u = \text{FIRST}(x)$
  - $X\alpha$  - obsah zásobníku ( $X$  je na vrcholu)
  - $\pi$  - výstup (levý rozklad)
- Počáteční konfigurace:  $(w, S\#, e)$
- Koncová konfigurace:  $(e, \#, \pi)$

Kam se schovaly stavy?

Programovací jazyky a překladače

78

## LL(1) gramatiky – Rozkladová tabulka



- $M: (\Gamma \cup \{\#\}) \times (\Sigma \cup \{e\}) \rightarrow \{\text{expand } i, \text{pop}, \text{accept}, \text{error}\}$
- **expand i:**  $(x, A\beta, \pi) :- (x, \alpha\beta, \pi.i)$ 
  - pravidlo  $p_i: A \rightarrow \alpha$
- **pop:**  $(ax, a\beta, \pi) :- (x, \beta, \pi)$
- **accept:**  $M[\#, e] = \text{accept}$ 
  - vstupní řetězec je přijat,  $\pi$  je levý rozklad
- **error:** syntaktická chyba

Programovací jazyky a překladače

79

## LL(1) gramatiky – Příklad (1)



- $S \rightarrow aAS$  (1)
- $S \rightarrow b$  (2)
- $A \rightarrow a$  (3)
- $A \rightarrow bSA$  (4)

	a	b	e
S	e1	e2	
A	e3	e4	
a	pop		
b		pop	
#			acc

Programovací jazyky a překladače

80



## LL(1) gramatiky – Příklad (2)



(abbab,S#,e)	:-[e1]	(abbab,aAS#,1)	:-[pop]
(bbab,AS#,1)	:-[e4]	(bbab,bSAS#,14)	:-[pop]
(bab,SAS#,14)	:-[e2]	(bab,bAS#,142)	:-[pop]
(ab,AS#,142)	:-[e3]	(ab,aS#,1423)	:-[pop]
(b,S#,1423)	:-[e2]	(b,b#,14232)	:-[pop]
(e,#,14232)	:-[acc]		

## LL(1) gramatiky – Konstrukce rozkladové tabulky



Je-li  $A \rightarrow \alpha$  i-té pravidlo, pak  $\forall a \in \text{FIRST}(\alpha)$ :

$M[A,a] = \text{expand } i$  ( $a \neq \epsilon$ )

Je-li  $\epsilon \in \text{FIRST}(\alpha)$ , pak  $\forall b \in \text{FOLLOW}(A)$

$M[A,b] = \text{expand } i$

$M[a,a] = \text{pop}$

$M[\#,e] = \text{accept}$

jinak  $M[X,a] = \text{error}$

## LL(1) gramatiky – Příklad



$S \rightarrow aAd \quad (1) \mid e \quad (2) \quad \text{FOLLOW}(S) = \{e\}$   
 $A \rightarrow bB \quad (3)$   
 $B \rightarrow cbB \quad (4) \mid e \quad (5) \quad \text{FOLLOW}(B) = \{d\}$

	a	b	c	d	e
S	e1				e2
A		e3			
B			e4	e5	
a	pop				
b		pop			
c			pop		
d				pop	
#					acc

Programovací jazyky a překladače

83

## LL(1) gramatiky – Převod na LL(1) gramatiku



- **Odstranění levé rekurze**

- $A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \dots \mid \beta_m \Rightarrow \text{FF neplatí}$

- $A \rightarrow \beta_1 A' \mid \dots \mid \beta_m A'$

- $A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_n A' \mid e$

- **Faktorizace** („vytknutí před závorku“)

- $A \rightarrow \beta\alpha_1 \mid \dots \mid \beta\alpha_n \Rightarrow \text{FF neplatí}$

- $A \rightarrow \beta A' \quad A' \rightarrow \alpha_1 \mid \dots \mid \alpha_n$

- **Eliminace pravidla** – dosazení pravých stran za neterminál

Programovací jazyky a překladače

84

## LL(1) gramatiky – Implementace LL(1) analyzátoru



- **Analýza s návraty**
  - Málo efektivní, Prolog
- **Analýza rekurzivním sestupem**
  - Vhodná pro „ruční“ implementaci
- **Nerekurzivní prediktivní analýza**
  - Využití rozkladové tabulky + interpretu
  - Vhodná pro generované analyzátory
- **Využití generátorů překladačů**
  - Obvykle umožňují nahlížet více symbolů než jeden

## LL(1) gramatiky – Rekurzivní sestup (1)



- Každý **neterminál** je reprezentován funkcí, která provádí jeho analýzu
- **Terminální symboly** jsou reprezentovány testem na jejich výskyt na vstupu následovaným čtením dalšího symbolu
- Překlad začíná voláním funkce reprezentující **startovací neterminál**

## LL(1) gramatiky – Rekurzivní sestup (2)



**Pravidlo**  $A \rightarrow X_1 X_2 \dots X_n$

$X \in \Sigma$  - volání expect(X)

$X \in N$  – volání X()

```
void A() {  
    // analýza X1  
    // analýza X2  
    ...  
    // analýza Xn  
}
```

Programovací jazyky a překladače

87

## LL(1) gramatiky – Rekurzivní sestup (3)



```
Symbol sym; // aktuální symbol
```

```
void expect(Symbol s)  
{  
    if( sym == s )  
        sym = lex(); // čti další symbol  
    else  
        error(); // syntaktická chyba  
}
```

Programovací jazyky a překladače

88

## LL(1) gramatiky – Příklad rekurzivního sestupu



$A \rightarrow xBy$

```
void A()
{
    expect(x);
    B();
    expect(y);
}
```

Programovací jazyky a překladače

89

## LL(1) gramatiky – Neterminál s více pravidly



- $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$

Výběr pravidla závisí na následujícím symbolu:

```
if( sym in SELECT(A,  $\alpha_i$ ) ) {
    // analýza řetězce  $\alpha_i$ 
}
```

- $\text{SELECT}(A, \alpha_i) =$

- $\text{FIRST}(\alpha_i)$   $e \notin \text{FIRST}(\alpha_i)$
- $\text{FOLLOW}(A) \cup (\text{FIRST}(\alpha_i) \setminus \{e\})$   $e \in \text{FIRST}(\alpha_i)$

Programovací jazyky a překladače

90

## LL(1) gramatiky – Příklad (1)



$E \rightarrow T E1$   
 $E1 \rightarrow + T E1 \mid e$   
 $T \rightarrow F T1$   
 $T1 \rightarrow * F T1 \mid e$   
 $F \rightarrow ( E ) \mid id$

## LL(1) gramatiky – Příklad (2)



- $FIRST(+ T E1) = \{+\}$   
 $FIRST(e) = \{e\}$   
 $FOLLOW(E1) = \{ \}, e\}$ 
  - e ve FOLLOW znamená eof
- $SELECT(A, + T E1) = \{+\}$   
 $SELECT(A, e) = \{ \}, e\}$

## LL(1) gramatiky – Příklad (3)



```
void E1()
{
    if( sym=='+' ) {
        expect('+');
        T();
        E1();
    } else if( sym=='\'' || sym==EOF ] ) {
        // prázdná pravá strana
    } else
        error();
}
```

Programovací jazyky a překladače

93

## LL(1) gramatiky – Nerekurzivní prediktivní analýza



- Syntaktický analyzátor řízený tabulkou rozkladovou tabulkou (zásobníkový automat)
- Rozkladová tabulka  $M[A,a]$
- Zásobník  $\text{push}(\alpha)$ ,  $\text{pop}()$ ,  $\text{top}()$ ,  $\text{init}()$
- Výstup  $\text{output}(i)$
- Zpracování chyb  $\text{error}()$

Programovací jazyky a překladače

94

## LL(1) gramatiky – Algoritmus interpretu (1)



```
init();
push($S);          // inicializace
a = lex();         // první symbol
do {
  X = top();
  if( X ∈ Σ ∪ { $ } ) // terminál nebo EOF($)

    if( X == a ) {
      pop();
      a = lex();
    } else
      error();
}
```

## LL(1) gramatiky – Algoritmus interpretu (2)



```
else if( M[X,a]==pi ) {
  pop(); // pi → Y1...Yk
  push(YkYk-1...Y1); // expanze
  output(i);
} else
  error();

} while( X != $ );
```



## LL(1) gramatiky – Zotavení po chybě (1)



- **Lexikální analýza**

- nesprávný formát čísla oprava
- neukončený řetězec oprava
- neznámý symbol přeskočení
- neukončená poznámka oprava

- **Syntaktická analýza**

- na úrovni symbolů přeskočení
- na úrovni vět oprava

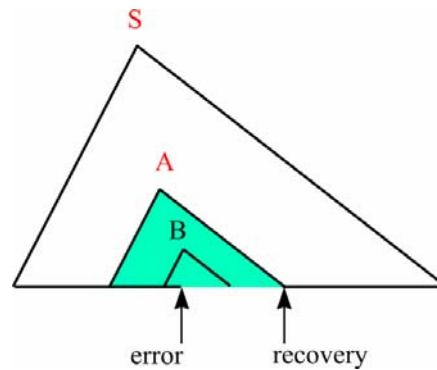
## LL(1) gramatiky – Zotavení po chybě (2)



- **Sémantická analýza**

- nedeklarovaný identifikátor oprava
- typová chyba oprava

## LL(1) gramatiky – Zotavení při analýze zhora dolů



Programovací jazyky a překladače

99

## LL(1) gramatiky – Postup při zotavení



Klíč = terminální symbol, umožňující spolehlivé určení místa v gramatice  
switch do if for == ? ...

- Nalezení klíče ve vstupní větě, přeskočení části textu
- 2) Přesun řízení na odpovídající místo v gramatice
- 3) Pokračování v analýze

Programovací jazyky a překladače

100

## LL(1) gramatiky – Problémy



- Nejednoznačnost klíčů
  - výskyt klíče v různých místech gramatiky
- Protichůdné požadavky
  - Malá množina klíčů
    - Roste délka přeskokovaného textu
  - Velká množina klíčů
    - Klesá spolehlivost zotavení
    - Zavlečené chyby

## LL(1) gramatiky – Metody zotavení



- Nerekurzivní zotavení s pevnou množinou klíčů
  - Množina klíčů = {“,”, “)”}
  - <příkaz> končí symbolem “,”
  - <výraz> končí symbolem “)”
  - if A= **then** A:=((+B); A:=A+B;
- Rekurzivní zotavení s pevnou množinou klíčů
  - Synchronizace na začátku konstrukce
    - “if” uvozuje <příkaz>
    - “(“ uvozuje <výraz>
  - if A= **then** A:=((+B); A:=A+B;
- Metoda s dynamicky budovanou množinou klíčů
  - Hartmannova metoda zotavení

## LL(1) gramatiky – Implementace Hartmannovy metody (1)



- Kdy se hlásí syntaktická chyba?
  - Funkce `expect` - na vstupu je neočekávaný symbol
  - Nelze vybrat pravidlo pro expanzi (symbol na vstupu není v  $\Phi(A, \alpha_i)$  pro žádné  $i$ )

## LL(1) gramatiky – Implementace Hartmannovy metody (2)



- Funkce reprezentující neterminály obdrží množinu klíčů jako parametr
  - `void E(Set c) { ... }`
- Při volání neterminálu nebo funkce `expect` se vypočte nová množina klíčů
  - `E( c ∪ { '+', '-' } )`  
`expect( '(', c ∪ { NUM, '(' } );`
- Funkce `check()` se volá na začátku neterminálu s více pravidly

## LL(1) gramatiky – Implementace Hartmannovy metody (3)



- $A \rightarrow X_1 X_2 \dots X_i X_{i+1} \dots X_k$ 
  - $c$  ... množina klíčů neterminálu  $A$
  - $c_i$  ... množina klíčů symbolu  $X_i$
- 1)  $c_i = c \cup \text{FOLLOW}(X_i)$   $X_i \in N$   
 $c_i = c$   $X_i \in \Sigma$
- 2)  $c_i = c \cup (\text{FIRST}(X_{i+1} \dots X_k) \setminus \{e\})$
- 3)  $c_i = c \cup ((\text{FIRST}(X_{i+1}) \cup \dots \cup \text{FIRST}(X_k)) \setminus \{e\})$

## Syntaxí řízený překlad - Překlad



- **Definice:** Překlad je relace  
TRAN:  $L_1 \rightarrow L_2$   
 $L_1 \subseteq \Sigma^*$   $\Sigma$  - vstupní abeceda  
 $L_2 \subseteq \Delta^*$   $\Delta$  - výstupní abeceda
- **Příklad**
  - $L_1$  – jazyk infixových výrazů
  - $L_2$  – jazyk postfixových výrazů

## Syntaxí řízený překlad - Překladová párová gramatika



- **Definice:** Překladová párová gramatika

$V = (N, \Sigma, \Delta, P, S)$

N - neterminály

$\Sigma$  - vstupní abeceda

$\Delta$  - výstupní abeceda

P – přepisovací pravidla

S – startovací neterminál

$P: A \rightarrow \alpha, \beta \quad \alpha \in (N \cup \Sigma)^*, \beta \in (N \cup \Delta)^*$

Neterminály z  $\beta$  jsou permutacemi neterminálů z  $\alpha$

Programovací jazyky a překladače

107

## Syntaxí řízený překlad - Příklad (1)



$E \rightarrow E+T, ET+$

$E \rightarrow T, T$

$T \rightarrow T^*F, TF^*$

$T \rightarrow F, F$

$F \rightarrow (E), E$

$F \rightarrow i, i$

$[E, E] \Rightarrow [E+T, ET+]$

$\Rightarrow [T+T, TT+]$

$\Rightarrow [F+T, FT+]$

$\Rightarrow [i+T, iT+]$

$\Rightarrow [i+T^*F, iTF^*+]$

$\Rightarrow [i+F^*F, FF^*+]$

$\Rightarrow [i+i^*F, iiF^*+]$

$\Rightarrow [i+i^*i, iii^*+]$

Programovací jazyky a překladače

108

## Syntaxí řízený překlad - Překladová gramatika



- Jsou-li neterminály ve vstupní i výstupní části pravé strany ve stejném pořadí, můžeme obě části spojit  
=> **překladová gramatika**
- Musíme ale odlišit vstupní a výstupní symboly

## Syntaxí řízený překlad - Příklad


$$\begin{aligned} E &\rightarrow E + T + \\ &\quad (E \rightarrow E + T \oplus) \\ E &\rightarrow T \\ T &\rightarrow T * F * \\ T &\rightarrow F \\ F &\rightarrow ( E ) \\ F &\rightarrow i i \end{aligned}$$
$$\begin{aligned} E &\Rightarrow E + T + \\ &\Rightarrow T + T + \\ &\Rightarrow F + T + \\ &\Rightarrow i i + T + \\ &\Rightarrow i i + T * F * + \\ &\Rightarrow i i + F * F * + \\ &\Rightarrow i i + i i * F * + \\ &\Rightarrow i i + i i * i i * + \end{aligned}$$

## Syntaxí řízený překlad - Homomorfismy



- Vstupní homomorfismus:
 
$$\begin{aligned} \iota(x) &= x, & x \in (N \cup \Sigma) \\ &= \varepsilon, & x \in \Delta \end{aligned}$$
- Výstupní homomorfismus:
 
$$\begin{aligned} o(x) &= x, & x \in (N \cup \Delta) \\ &= \varepsilon, & x \in \Sigma \end{aligned}$$
- Příklad
  - $\iota(E + T +) = E + T$   
 $o(E + T +) = E T +$
  - $\iota(i + i i^* i i^* +) = i + i^* i$   
 $o(i + i i^* i i^* +) = i i i^* +$

Programovací jazyky a překladače

111

## Syntaxí řízený překlad - Překlad



- **Definice:** Překlad generovaný překladovou gramatikou:

$$T(G) = \{(x,y) \mid \begin{aligned} &S \Rightarrow^* z, \\ &z \in (\Delta \cup \Sigma)^*, \\ &x = \iota(z), \\ &y = o(z) \end{aligned} \quad \begin{array}{l} \text{vstupní věta} \\ \text{výstupní věta} \end{array}\}$$

Programovací jazyky a překladače

112



## Syntaxí řízený překlad - Atributovaný překlad



- Rozšíření bezkontextové gramatiky o kontextové vlastnosti  
→ „**gramatika s parametry**“
- Jednotlivé symboly mohou mít přiřazeny parametry – **atributy**
- Atributy se předávají podobně jako vstupní a výstupní argumenty funkcí

## Syntaxí řízený překlad - Atributová překladová gramatika



- **Definice:** Atributová *překladová* gramatika  
 $G_{AT} = (G_T, A, F)$   
 $G_T$  – překladová gramatika  
( $\Delta = \emptyset$  -> **atributová gramatika**)  
 $A$  – množina atributů  
 $F$  – sémantická pravidla (pravidla pro výpočet hodnot atributů)

## Syntaxí řízený překlad - Množina atributů



- $X \in (\Delta \cup \Sigma \cup N)$ 
  - $I(X)$  - množina **dědičných** atributů
  - $S(X)$  - množina **syntetizovaných** atributů
- Dědičné atributy startovacího neterminálu jsou zadány předem.
- Syntetizované atributy terminálních symbolů jsou zadány předem (lexikální analyzátor)

## Syntaxí řízený překlad - Sémantická pravidla (1)



$p_r: X_0 \rightarrow X_1 X_2 \dots X_n \quad X_0 \in N, X_i \in (N \cup \Delta \cup \Sigma)$

a)  $d = f_r^{d,k}(a_1, a_2, \dots, a_n) \quad d \in I(X_k), 1 \leq k \leq n$

b)  $s = f_r^{s,0}(a_1, a_2, \dots, a_n) \quad s \in S(X_0)$

c)  $s = f_r^{s,k}(a_1, a_2, \dots, a_n) \quad s \in S(X_k), X_k \in \Delta$

- a), b)  $a_i$  je atribut symbolu  $X_j, 0 \leq j \leq n$
- c)  $a_i$  je dědičný atr. symbolu  $X_r, X_r \in \Delta$

## Syntaxí řízený překlad - Sémantická pravidla (2)



- Výklad předchozích definic:
  - a) Dědičné atributy symbolu na pravé straně pravidla
  - b) Syntetizované atributy neterminálního symbolu na levé straně pravidla
  - c) Syntetizované atributy výstupních symbolů na pravé straně pravidla

## Syntaxí řízený překlad - Syntaxí řízená definice



- sémantické funkce v atributové gramatice nemohou mít vedlejší efekty
  - vstup/výstup (např. generovaného kódu)
  - globální proměnné (např. tabulka symbolů)
- *syntaxí řízená definice* – bez omezení

## Syntaxí řízený překlad - Syntaxí řízené definice a překladová schémata



- Obecně existují dvě notace, jak můžeme připojit sémantické akce k pravidlům gramatiky.
  - Syntaxí řízené definice
    - Základní idea je, že jednotlivým pravidlům přiřadíme množinu sémantických akcí.
    - Specifikace překladu na vysoké úrovni abstrakce.
    - Ukrywají mnoho implementačních detailů jako například pořadí vyhodnocování pravidel.
  - Překladová schémata
    - Určuje pořadí vyhodnocení jednotlivých sémantických pravidel.
    - Umožňuje definovat nějaké implementační detaily

## Syntaxí řízený překlad - Překladové schéma



- Syntaxí řízená definice se sémantickými akcemi umístěnými kdekoli na pravé straně pravidla
- Posloupnost sémantických akcí je přesně definována

## Syntaxí řízený překlad - Návrh překladového schématu



- Dědičný atribut symbolu na pravé straně vypočte akce umístěna před ním.
- Akce nesmí používat syntetizované atributy symbolů vpravo od ní.
- Hodnota syntetizovaného atributu neterminálu na levé straně pravidla se vypočte až po určení všech atributů, na které se odkazuje - obvykle až na konci pravidla

## Syntaxí řízený překlad - L - atributová definice



### L-atributová definice

$$A \rightarrow X_1 \dots X_n$$

Dědičné atributy symbolu  $X_j$  závisejí jen na:

- attributech symbolů  $X_1, \dots, X_{j-1}$   
(tj. symbolech z téhož pravidla vlevo od symbolu  $X_j$ )
- dědičných attributech symbolu  $A$   
(tj. symbolu na levé straně pravidla)
- **Použití: LL(1) překlad – jasně definované pořadí pro vyhodnocení sémantických pravidel.**

## Syntaxí řízený překlad - Příklad na výpočet hodnoty výrazu (1)



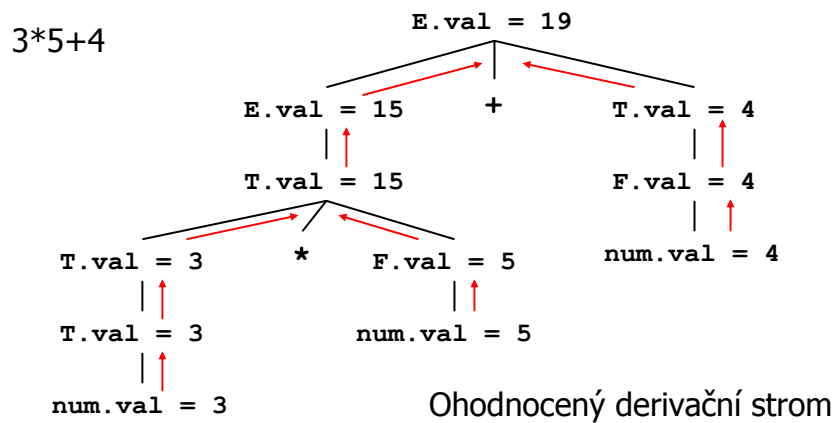
- Syntetizované atributy:
  - E.val - hodnota výrazu
  - T.val - hodnota podvýrazu
  - F.val - hodnota operandu
  
- num.lexval – hodnota číselné konstanty

## Syntaxí řízený překlad - Příklad na výpočet hodnoty výrazu (2)



- |                          |                            |
|--------------------------|----------------------------|
| • Gramatika              | • Sémantická pravidla      |
| • $E \rightarrow E' + T$ | • $E.val = E'.val + T.val$ |
| • $E \rightarrow T$      | • $E.val = T.val$          |
| • $T \rightarrow T' * F$ | • $T.val = T'.val * F.val$ |
| • $T \rightarrow F$      | • $T.val = F.val$          |
| • $F \rightarrow (E)$    | • $F.val = E.val$          |
| • $F \rightarrow num$    | • $F.val = num.lexval$     |

## Syntaxí řízený překlad - Příklad na výpočet hodnoty výrazu (3)



Programovací jazyky a překladače

125

## Syntaxí řízený překlad - Příklad deklarace proměnných (1)



- |                            |  |
|----------------------------|--|
| (1) $D \rightarrow T L$    | $L.type := T.type$                           |
| (2) $T \rightarrow int$    | $T.type := integer$                          |
| (3) $T \rightarrow real$   | $T.type := real$                             |
| (4) $L \rightarrow L', id$ | $L'.type := L.type$<br>$dcl(id.ptr, L.type)$ |
| (5) $L \rightarrow id$     | $dcl(id.ptr, L.type)$                        |

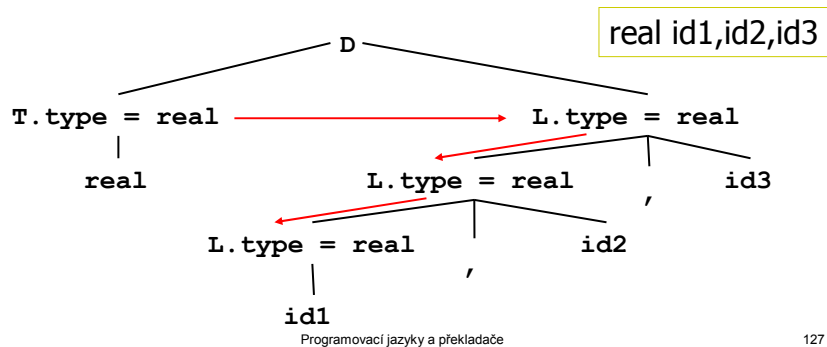
Programovací jazyky a překladače

126

## Syntaxí řízený překlad - Příklad deklaráce proměnných (1)



- Dědičné atributy: L.type
- Syntetizované atributy: T.type, id.ptr



## Syntaxí řízený překlad - Příklad syntaxí řízených definic



$E \rightarrow T R$

$R \rightarrow + T \{ \text{print}(' + '); \} R'$   
 $\quad \mid \epsilon$

$T \rightarrow \text{num} \{ \text{print}(\text{num.val}); \}$



## Syntaxí řízený překlad - Překlad při analýze rekurzivním sestupem (1)



- Syntaktický zásobník - řízení překladu
- Atributový zásobník - ukládání atributů
  
- Atributy
  - parametry podprogramů  $(N \cup \Sigma)$
  - globální proměnné  $\Delta$

## Syntaxí řízený překlad - Překlad při analýze rekurzivním sestupem (2)



- Atributy
  - dědičné - vstupní parametry
  - syntetizované - výstupní parametry
- Sémantické akce
  - Přímo na odpovídajících místech v pod programech pro analýzu pravidel
- Hodnoty syntetizovaných atributů **musí** být definovány! (e-pravidla, zotavení)

## Syntaxí řízený překlad - Příklad implementace rekurzivním sestupem (1)



```
E -> T {R.i := T.val} R {E.val := R.s}
R -> addop T
      {if addop.op=add then
        R1.i := R.i+T.val
        else R1.i := R.i-T.val }
      R1 {R.s := R1.s}
      | ε {R.s := R.i}
T -> num {T.val := num.val}
```

## Syntaxí řízený překlad - Příklad implementace rekurzivním sestupem (2)



- addop.op ... var lexop:char; '+'/'-'  
num.val ... lexval:integer;
- R.i - dědičný atribut (levý operand)  
R.s - dočasný výsledek, synt. atribut  
T.val - syntetizovaný attribute

## Syntaxí řízený překlad - Příklad implementace rekurzivním sestupem (3)



```
void E(int& val) {  
    int val1;  
    T(val1);  
    R(val1, val);  
}
```

## Syntaxí řízený překlad - Příklad implementace rekurzivním sestupem (4)



```
void R(int i, int& s) {  
    if( sym == '+' || sym == '-' ) {  
        char op=lexop;  
        int val;  
        T(val);  
        if( op=='+' ) R(i+val,s)  
            else R(i-val,s);  
    } else  
        s = i;  
}
```

## Syntaxí řízený překlad - Příklad implementace rekurzivním sestupem (5)



```
void T(int& val) {
    if( sym == num ) {
        val=lexval; // nestačí expect(num)
        lex();
    } else {
        error(); // po zotavení musí
        val=0;    // být val definováno
    }
}
```

## JavaCC - Konstruktor JavaCC



- Generuje lexikální a syntaktický analyzátor v jazyce Java
- Lexikální analýza používá regulární výrazy
- Syntaktická analýza
  - rekurzivní sestup
  - LL(k) gramatika
  - Extended BNF - EBNF (rozšířená BNF s regulárními operátory)

## JavaCC - Formát specifikace



```
/* volby generátoru */
options {
    IGNORE_CASE = true; DEBUG_PARSER = true;
}
/* třída analyzátoru */
PARSER_BEGIN(Calc)
public class Calc {
    public static void main(String args[])
        throws ParseException
    { Calc parser = new Calc(System.in);
      parser.expr();
    }
}
PARSER_END(Calc)
```

## JavaCC - Formát specifikace



```
/* lexikální analyzátor */
SKIP :
{ " " | "\r" | "\t" }
TOKEN :
{ < EOL: "\n" > }
TOKEN :
{ <ADD: "+"> | <SUB: "-"> |
  <MUL: "*"> | <DIV: "/"> | <MOD: "mod"> }
TOKEN :
{ <CONSTANT: ( <DIGIT> )+ > |
  <#DIGIT: ["0" - "9"]> }
TOKEN :
{ < SEMICOLON: ";" > }
```

## JavaCC - Formát specifikace



```
/* syntaktický analyzátor */
void expr() : { }
{ term() (( "+" | "-" ) term())* }

void term() : { }
{ factor() (("*" | "/" | "mod") factor())* }

void factor() : { }
{ <CONSTANT> | "(" expr() ")" }
```

## JavaCC – Spuštění JavaCC



```
D:\home\JavaCC-Book\adder>javacc Calc.jj
Java Compiler Compiler Version 2.1 (Parser Generator)
Copyright (c) 1996-2001 Sun Microsystems, Inc.
Copyright (c) 1997-2001 WebGain, Inc.
(type "javacc" with no arguments for help)
Reading from file adder.jj . . .
File "TokenMgrError.java" does not exist. Will create one.
File "ParseException.java" does not exist. Will create one.
File "Token.java" does not exist. Will create one.
File "SimpleCharStream.java" does not exist. Will create one.
Parser generated successfully.
This generates seven Java classes, each in its own
```

- Hlavní soubor bude Calc.java
- Jde o běžný zdrojový soubor v Javě

## JavaCC – Hodnota lexikálního symbolu



```
int Start() throws NumberFormatException :
{
    Token t ;
    int i ;
    int value ;
}{
    t = <NUMBER>
    { i = Integer.parseInt( t.image ) ;
      value = i ; }
    ( <PLUS> t = <NUMBER>
      { i = Integer.parseInt( t.image ) ; }
      { value += i ; }
    )*
    <EOF> { return value ; }
}
```

Programovací jazyky a překladače

141

## Tabulka symbolů - Tabulka symbolů



- Reprezentace pojmenovaných entit
  - explicitně pojmenovaných uživatelem
  - implicitně pojmenovaných – standardní entity (typy, funkce, třídy, ...), dočasné proměnné
- Účel:
  - řešení kontextových vazeb (deklarace -> použití)
  - typová kontrola
  - ukládání informací pro generování kódu

Programovací jazyky a překladače

142

## Tabulka symbolů - Tabulka symbolů



- Interakce s lexikálním analyzátořem
  - ukládání identifikátorů během lexikální analýzy
  - využití kontextových informací
    - Expr -> IdVar Index
    - | IdProc Args
    - | IdCon
    - ...
  - PASCAL: **P**(x,y) **kontextově závislá syntaxe**  
**Write**(x:3,y:2:5)

## Tabulka symbolů - Funkce tabulky symbolů



- **Operace:**
  - init - inicializace
  - insert - vkládání
  - lookup - vyhledávání (častější)
- **Inicializace**
  - vytvoření prázdné tabulky
  - naplnění implicitními deklaracemi



## Tabulka symbolů - Funkce tabulky symbolů



- **Vkládání** – nejprve vyhledá klíč v tabulce
  - není nalezen → vytvoří se nová položka
  - nalezen → obvykle chyba
    - deklarace / definice
    - implicitní deklarace (návěští a funkce v C)
- **Vyhledávání** – vyhledá klíč v tabulce
  - nalezen → vrátí odpovídající entitu
  - nenalezen → obvykle ohlásí chybu

Programovací jazyky a překladače

145

## Tabulka symbolů - Implementace tabulky symbolů (1)



- Neseřazené tabulky  $O(n)$ 
  - jen pro malý počet prvků
- Seřazené tabulky s binárním vyhledáváním  $O(\log_2 n)$ 
  - pro statické tabulky (např. klíčová slova)
- Vyhledávací stromy  $O(n) - O(\log_2 n)$ 
  - doba vyhledávání závisí na vyvážení stromu
  - časová náročnost vkládání není kritická
  - optimálně vyvážené stromy
    - příliš komplikované
  - suboptimální řešení
    - např. AVL stromy

Programovací jazyky a překladače

146

## Tabulka symbolů - Implementace tabulky symbolů (2)



- Tabulky s rozptýlenými položkami  $O(1)$ 
  - mapovací funkce: *klíč* → *index*
  - řešení kolizí
    - seznamy/stromy synonym
    - rozptylovací funkce: *index* → *index*
  - rychlost závisí na zaplnění tabulky
  - problematický průchod položkami podle klíče
  - obtížně se řeší přetečení tabulky

## Tabulka symbolů - Blokově strukturovaná tabulka (1)



- Vhodná pro jazyky se zanořenými deklaracemi (hierarchické struktury)
- Řeší rozsah platnosti a viditelnost jména v zanořených blocích
- Nové operace:
  - otevření rozsahu platnosti (**open**)
  - uzavření rozsahu platnosti (**close**)

## Tabulka symbolů - **Blokově strukturovaná tabulka (2)**



- Vkládání
  - pracuje pouze s aktuální úrovní tabulky
  - jména na vyšších úrovních se neuvažují
- Vyhledávání
  - klíč se vyhledává nejprve v aktuální úrovni, pokud se nenajde, hledá o úroveň výš
  - neúspěch se hlásí až po prohledání nejvyšší úrovně (globální jména)
  - explicitní přístup na globální úroveň ::x

## Tabulka symbolů - **Implementace blokově strukturované tabulky symbolů**



- Založena na některé z metod pro nestrukturovanou tabulku
  - vyhledávací stromy
  - tabulky s rozptýlenými položkami
- Přirozenou datovou strukturou pro reprezentaci hierarchie je **ZÁSOBNÍK**
  - Úrovně se nemohou překrývat

## Tabulka symbolů - Příklad



	jméno entity	další atributy
8	y	
7	delta	
6	max	
5	num	
4	x	
3	fun	
2	y	
1	x	

```
int x,y;
double fun(x,num)
{
    int max;
    ...
    {
        double delta,y;
        ...
    }
}
```

index  
bloku

Programovací jazyky a překladače

151

## Struktura programu v době běhu - Obsah



- Vztah mezi zdrojovým programem a činností přeloženého programu
  - reprezentace dat
  - správa paměti
  - aktivace podprogramů
    - parametry
    - rekurze

Programovací jazyky a překladače

152

## Struktura programu v době běhu - Systém řízení běhu programu (1)



- Inicializace programu
  - převzetí parametrů od systému
  - alokace paměti pro data
  - otevření standardních souborů (Pascal)
- Finalizace programu
  - uzavření souborů
  - uvolnění paměti
  - předání stavového kódu

## Struktura programu v době běhu - Systém řízení běhu programu (2)



- Zpracování chyb
  - přerušení a výjimky (definované uživatelem, systémové a aplikační chyby, ...)
  - automatické kontroly (přetečení zásobníku, ukazatele, indexy, parametry, ...)
- Dynamické přidělování paměti
  - zásobník
  - volná paměť
- Komunikace s operačním systémem
  - soubory, čas, ...
- Volací a návratové posloupnosti
- Podpora pro ladění programu

## Struktura programu v době běhu – Podprogramy



- Procedury
  - jméno -> příkaz
- Funkce
  - jméno -> výraz

```
void main() {  
    int fib(int n) {  
        if( n < 2 ) return 1;  
        return fib(n-2) + fib(n-1);  
    }  
    void print(int n) {  
        for(int k = 1; k <= n; k++)  
            printf("%d: %d", k, fib(k));  
    }  
    int max;  
    scanf("%d", &max);  
    print(max);  
}
```

Programovací jazyky a překladače

155

## Struktura programu v době běhu – Definice podprogramu



- Jméno
- Formální parametry (parametry)
  - jméno
  - typ
  - způsob předávání
- Tělo

Programovací jazyky a překladače

156

## Struktura programu v době běhu – **Aktivace podprogramu**



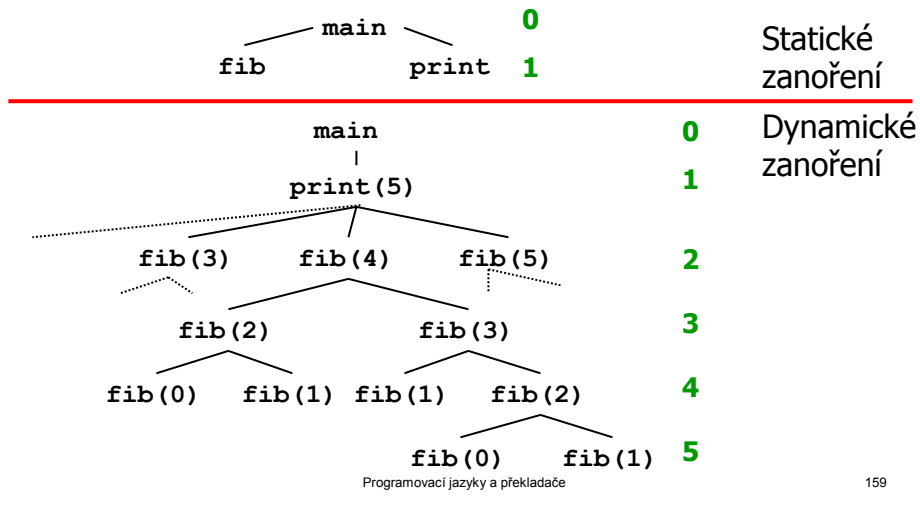
- Skutečné parametry (argumenty)
- Doba života aktivace
  - od zavolání podprogramu po návrat zpět
  - obvykle se nepřekrývají nebo jsou do sebe vnořeny (výjimka: vlákna, procesy)
- Rekurzivní podprogramy
  - nová aktivace se může spustit před ukončením předcházející

## Struktura programu v době běhu – **Úroveň zanoření**



- Statická úroveň zanoření
  - je definována zdrojovým programem
  - C, Java – pouze jedna úroveň
- Dynamická úroveň zanoření
  - je definována zanořováním aktivací
  - kolik aktivací je současně rozpracováno
- Úroveň zanoření je dána vzdáleností uzlu od kořene stromu

## Struktura programu v době běhu – Příklad



159

## Struktura programu v době běhu – Lokální data aktivace (1)



- Aktivační záznam
  - lokální proměnné (+dočasné proměnné)
  - návratová adresa
  - uschovaný obsah registrů
- Jazyky bez rekurzivních podprogramů
  - aktivační záznamy mohou být alokovány staticky v době překladu (FORTRAN)
- Jazyky s rekurzivními podprogramy
  - zásobník aktivačních záznamů
  - řídicí zásobník
    - obsahuje všechny aktivační záznamy od kořene stromu aktivací až po aktivní uzel (podprogram)

Programovací jazyky a překladače

160

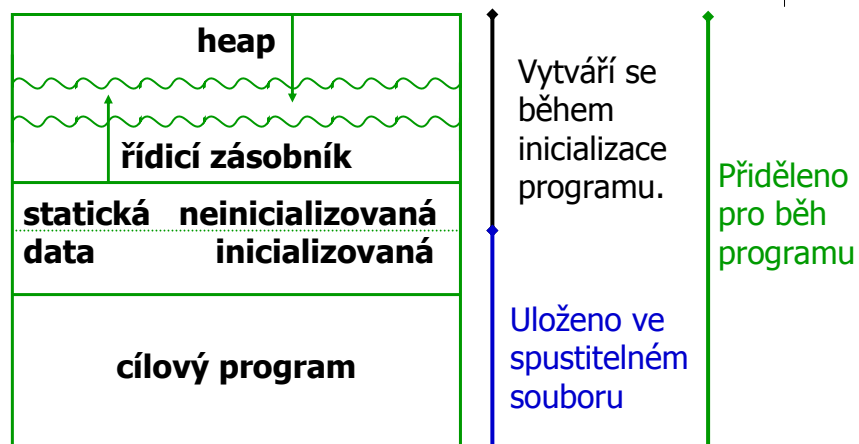


## Struktura programu v době běhu – **Struktura paměti**



- Cílový program **program, konstanty**
- Statická data **globální a statické proměnné**
- Řídicí zásobník **aktivační záznamy (lokální proměnné)**
- Volná paměť **dynamicky alokovaná paměť (hromada – heap)**

## Struktura programu v době běhu – **Schéma obsazení paměti**



## Struktura programu v době běhu – Přidělování paměti pro aktivační záznamy (1)



- Statické přidělování paměti
  - adresy všech proměnných jsou známy v době překladu
  - lokální proměnné mohou přežít volání programu (viz static v C)
  - počet a velikost prvků musíme znát v době překladu
  - omezená možnost rekurze (sdílení proměnných)
    - obsahy předchozích aktivačních záznamů můžeme odkládat na zásobník

## Struktura programu v době běhu – Přidělování paměti pro aktivační záznamy (2)



- Přidělování na zásobníku
  - vhodné pro jazyky s rekurzí
  - paměť se přidělí v okamžiku volání
  - uvolnění paměti proběhne při návratu
  - relativní adresování proměnných – offsety jsou známy v době překladu
  - neznáme-li velikost parametrů
    - > **deskriptor [adresa, délka]**
      - Příklad: řetězce, otevřená pole

## Struktura programu v době běhu – Přidělování paměti pro aktivační záznamy (3)



- Přidělování z volné paměti
  - lokální proměnné mohou přežít aktivaci podprogramu, resp. aktivace volaného podprogramu může přežít aktivaci volajícího podprogramu -> **nelze použít zásobník**
  - alokace paměti při inicializaci aktivace, finalizace až ve chvíli, kdy není potřebná
  - implementace paralelních jazyků

## Struktura programu v době běhu – Volací a návratové posloupnosti



- Definice adresy aktivačního záznamu  
Uložení stavu (registry, příznaky, návratová adresa)  
Inicializace lokálních dat
- ...
- Uložení výsledku  
Obnovení předchozího stavu  
Obnovení adresy aktivačního záznamu volajícího  
Návrat za místo volání

## Struktura programu v době běhu – Reprezentace datových typů

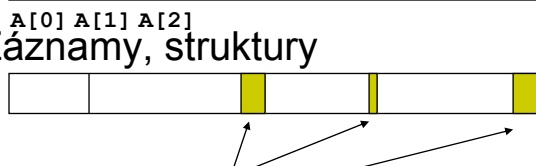


- Primitivní datové typy:
  - char, int, double
  - výčet, interval – stejně jako int

- Pole



- Záznamy, struktury



**zarovnání**

Programovací jazyky a překladače

167

## Vnitřní reprezentace - Možnosti překladačů



- Interpretace
  - Okamžité provádění programu
- Překlad do instrukcí procesoru
  - Závislost na konkrétním typu procesoru
- **Překlad do vnitřní reprezentace**
  - Následuje interpretace nebo překlad do instrukcí procesoru

Programovací jazyky a překladače

168

## Vnitřní reprezentace - Výhody překladačů do vnitřní reprezentace



- Strukturalizace překladače
- Mnohem jednodušší přenos na více typů procesorů
- Možnost optimalizace na úrovni vnitřní reprezentace
  - strojově nezávislé metody

## Vnitřní reprezentace - Formáty vnitřní reprezentace programu



- **Grafová reprezentace**
- **Zásobníkový kód**
- **Třídresový kód**

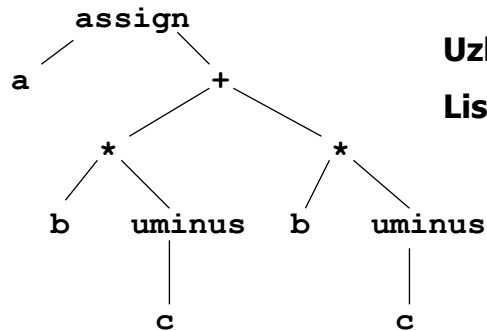
## Vnitřní reprezentace - Grafová reprezentace



- Abstraktní syntaktický strom (AST)

$a := b * (-c) + b * (-c)$

$c) + b * (-c)$



Uzly - operátory

Listy - operandy

Programovací jazyky a překladače

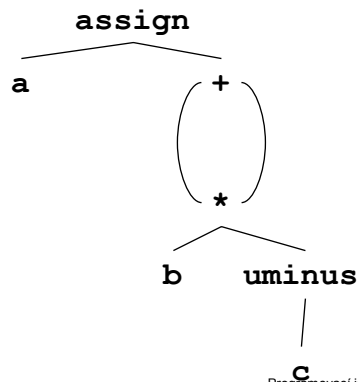
171

## Vnitřní reprezentace - Grafová reprezentace



- DAG (Directed Acyclic Graph)

$a := b * (-c) + b * (-c)$



Společné podvýrazy jako sdílené uzly

Používá se pro optimalizaci

Programovací jazyky a překladače

172

## Vnitřní reprezentace - Grafová reprezentace



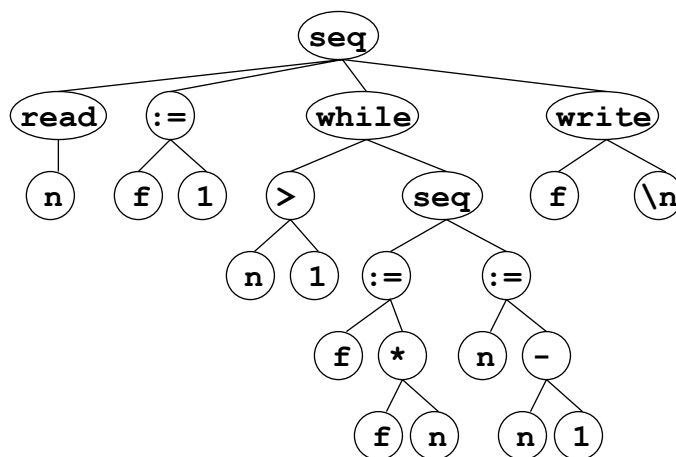
- Obecný grafový model
  - OO modelování a návrh

```
var n,f:integer;  
begin  read n;  
      f:=1;  
      while n>1 do begin  
        f:=f*n; n:=n-1  
      end;  
      write f,"\n"  
end.
```

Programovací jazyky a překladače

173

## Vnitřní reprezentace - Grafová reprezentace



Programovací jazyky a překladače

174

## Vnitřní reprezentace - Zásobníkový kód



- Postfixová notace

`a b c uminus * b c uminus * + assign`

- výsledek průchodu AST nebo DAG typu post-order
- uzel následuje bezprostředně za svými následníky
- žádný formální rozdíl mezi operandy a operátory

## Vnitřní reprezentace - Zásobníkový kód



- Virtuální zásobníkový procesor

VAR b	(b)	vrchol zás.
VAR c	(b) (c)	
INV	(b) (-c)	
MUL	(b*-c)	
VAR b	(b*-c)(b)	
VAR c	(b*-c)(b)(c)	
INV	(b*-c)(b)(-c)	
MUL	(b*-c)(b*-c)	
ADD	(b*-c+b*-c)	
ASG a		



## Vnitřní reprezentace - Zásobníkový kód



- Virtuální zásobníkový procesor
  - virtuální počítač s pamětí a zásobníkem
  - P-kód (Wirth)
    - přenositelný mezikód pro Pascal
    - specializované procesory
  - Java Virtual Machine
  - MSIL – Microsoft Intermediate Language (.NET)

## Vnitřní reprezentace - Tříadresový kód



- **$x := y \text{ op } z$** 
  - $x$  - (dočasná) proměnná
  - $y, z$  - (dočasné) proměnné, konstanty
- Operandů nemohou být výrazy
  - rozklad na primitivní výrazy s dočasnými proměnnými
- Explicitní odkazy na operandů
  - možnost přesouvání příkazů při optimalizaci

## Vnitřní reprezentace - Příkazy tříadresového kódu



- $x := y \text{ op } z$                        $\text{op}$  – binární operátor
- $x := \text{op } y$                                $\text{op}$  - unární operátor  
(-, not, typová konverze, ...)
  
- $x := y$
- goto L                                      nepodmíněný skok
- if x rel op y goto L                      podmíněný skok
  
- $x := y[j]$   
   $x[j] := y$                                   indexování
- $x := \&y$                                   reference (získání adresy)
  
- $x := *y$   
   $*x := y$                                   dereference ukazatele

## Vnitřní reprezentace - Příklad



- $a := b * (-c) + b * (-c)$
  
- $t1 := -c$                                        $t1 := -c$   
   $t2 := b * t1$                                    $t2 := b * t1$   
   $t3 := -c$                                        $t5 := t2 + t2$   
   $t4 := b * t3$                                    $a := t5$   
   $t5 := t2 + t4$   
   $a := t5$
  
- syntaktický strom      DAG
  
- dočasné proměnné = (vnitřní) uzly stromu

## Vnitřní reprezentace - Příklad – překlad výrazů (1)



- Atributy
  - *id.name* – jméno identifikátoru
  - *E.place* – proměnná obsahující hodnotu E
- Pomocné funkce
  - *newtemp* – vytvoří novou proměnnou
  - *lookup* – vyhledá proměnnou v tabulce
  - *emit* – generuje instrukci
  - *error* – hlášení chyby

## Vnitřní reprezentace - Příklad – překlad výrazů (2)



- $S \rightarrow id := E$ 

```
{ p = lookup(id.name);  
  if( p != null )  
    emit(p, ':=' ,E.place);  
  else error(); }
```
- $E \rightarrow E_1 + E_2$ 

```
{ E.place = newtemp();  
  emit(E.place, ':=' ,  
    E_1.place, '+', E_2.place); }
```
- $E \rightarrow E_1 * E_2$ 

```
{ E.place := newtemp();  
  emit(E.place, ':=' ,  
    E_1.place, '*', E_2.place); }
```

## Vnitřní reprezentace - Příklad – překlad výrazů (3)



- $E \rightarrow - E_1$ 

```
{ E.place = newtemp();  
  emit(E.place, ':=' ,  
        'uminus', E_1.place); }
```
- $E \rightarrow (E_1)$ 

```
{ E.place = E_1.place; }
```
- $E \rightarrow id$ 

```
{ p := lookup(id.name);  
  if( p != null )  
    E.place := p;  
  else error(); }
```

Programovací jazyky a překladače

183

## Vnitřní reprezentace - Překlad logických výrazů (1)



- $E \rightarrow E \text{ or } E$ 
  - |  $E \text{ and } E$
  - |  $(E)$
  - | **not**  $E$
  - |  $id \text{ relop } id$
  - | **true**
  - | **false**

Programovací jazyky a překladače

184

## Vnitřní reprezentace - Překlad logických výrazů (2)



- Reprezentace logických hodnot celými čísly:  
**true=1, false=0**

- **a or b and not c**

```
t1 := not c
t2 := b and t1
t3 := a or t2
```

**x<y**

```
1: if x<y goto 4
2: t1 := 0
3: goto 5
4: t1 := 1
5: ...
```

Programovací jazyky a překladače

185

## Vnitřní reprezentace - Překlad logických výrazů (3)



- Zkrácené vyhodnocení
  - reprezentace logických hodnot pozicí v programu

- **a<b or c<d and e<f**

```
if a<b goto Ltrue
goto L1
L1: if c<d goto L2
goto Lfalse
L2: if e<f goto Ltrue
goto Lfalse
```

Programovací jazyky a překladače

186

## Vnitřní reprezentace - Překlad řídicích příkazů (1)



- Instrukce zásobníkového kódu:
  - LBL – definice návěští pro skok
  - JMP L – nepodmíněný skok na návěští L
  - FJP L – podmíněný skok na návěští L, pokud je na vrcholu zásobníku False
- Pomocné funkce:
  - getLbl() – vygeneruje nové číslo návěští

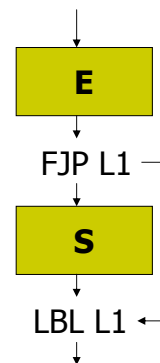
Programovací jazyky a překladače

187

## Vnitřní reprezentace - Překlad příkazu IF (1)



- $S \rightarrow \text{if} ( E ) S$ 
  - $\langle E \rangle$
  - FJP L1
  - $\langle S \rangle$
  - LBL L1
- $S \rightarrow \text{if} ( E ) \{ FJP \} S \{ LBL \}$   
 $FJP.l = LBL.l = \text{getLbl}();$  ↓



Programovací jazyky a překladače

188

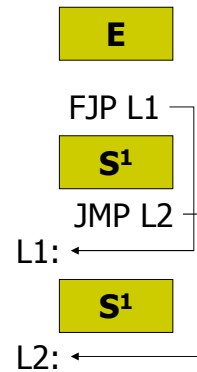
## Vnitřní reprezentace - Překlad příkazu IF (2)



- $S \rightarrow \text{if} ( E ) S^1 \text{ else } S^2$

```

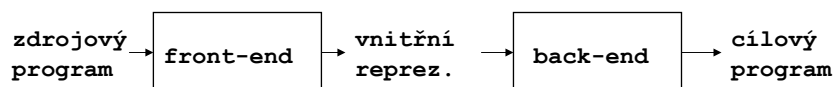
S -> X {LBL}
    LBL.I = X.I
S -> X {JMP} {LBL1} else S {LBL2}
    JMP.I = LBL2.I = getLbl();
    LBL1.I = X.I
X -> if (E) {FJP} S
    FJP.I = getLbl()
    
```



Programovací jazyky a překladače

189

## Optimalizace – Možnosti optimalizace



**uživatel**  
změna algoritmu  
transformace  
profilování

**překladač**  
strojově  
nezávislé  
optimalizace

**překladač**  
strojově  
závislé  
optimalizace

„Optimalizace“ = transformace programu směřující ke zlepšení jeho vlastností za běhu (rychlost, prostor)

Programovací jazyky a překladače

190

## Optimalizace – Možnosti optimalizace



- Optimalizace zdrojového programu
  - často největší vliv – např. změna algoritmu
  - možnost profilování programu
- Strojově nezávislé optimalizace
  - na úrovni vnitřní reprezentace
- Strojově závislé optimalizace
  - na úrovni cílového programu
  - co nejlepší využití instrukční sady procesoru a jeho zdrojů (registry, paměť, ...)

## Optimalizace – Kritéria pro transformace



- musí zachovat význam programu
  - "bezpečný" přístup --- raději neoptimalizovat!
- musí program v průměru významně zrychlit
  - někdy nám jde o velikost paměti
  - v některých případech se může efektivita programu i zhoršit
- musí stát za vynaložené úsilí
  - vyplatí se pro často spouštěné programy



## Optimalizace – Volba vnitřní reprezentace programu pro optimalizaci



- Zásobníkový kód
  - obtížná změna struktury programu
- **Třídresový kód**
  - explicitně vyjádřené závislosti
  - blízký architektuře RISC i klasickým architektuřám procesorů
- Grafová reprezentace
  - grafové transformace, přepisovací systémy

## Optimalizace – Organizace optimalizujícího překladače



- Analýza toku řízení
  - rozdělení programu na *základní bloky*
  - vytvoření *grafu toku řízení*
- Analýza toku dat
  - živé proměnné
  - dosahující definice
  - dostupné výrazy
- Transformace

## Optimalizace – Základní blok



*Nejdelší posloupnost příkazů s jedním vstupem a jedním výstupem*

- pouze poslední příkaz může být skok
- cílovým místem skoků může být pouze první příkaz bloku
- můžeme jej považovat za jedinou instrukci
- určuje meze lokálních optimalizací
- snížení množství uchovávaných dat
  - např. 16B/instrukci + 8KB/data pro optimalizaci
  - lze ukládat jednou pro základní blok a ostatní dopočítat

## Optimalizace – Typy optimalizací



- Lokální optimalizace
  - na úrovni *základního bloku*
  - nevyžaduje znalost kontextu
- Globální optimalizace
  - v rámci podprogramů
  - v rámci programu (interprocedurální)

## Optimalizace – Příklady lokálních optimalizací (1)



- odstranění společných podvýrazů  
`a[i,j] = a[i,j]+1;`
- vyhodnocení konstantních výrazů  
`a = 2 * 3 + 5;`
- šíření kopií  
`f = g;`
- šíření konstant  
`f = 2 * 3; if( f > 10 ) ...`

## Optimalizace – Příklady lokálních optimalizací (2)



- odstranění mrtvého kódu  
`debug = false; if (debug) printf("...");  
goto L; x = 10; L: ...`
- algebraické transformace  
`x * 1 + 0       => x`
- redukce síly operací  
`x ** 2           => x * x  
5 * x           => (x << 2) + x  
length(x+y)     => length(x)+length(y)`

## Optimalizace – Optimalizace cyklů



- přesun kódu mimo cyklus
  - invarianty cyklu lze přesunout před tělo cyklu
- redukce indukčních proměnných
  - generování posloupnosti hodnot
  - např. indexování pole proměnnou cyklu => použití ukazatele na aktuální prvek
- PROBLÉM: detekce cyklů

## Optimalizace – Globální optimalizace programu



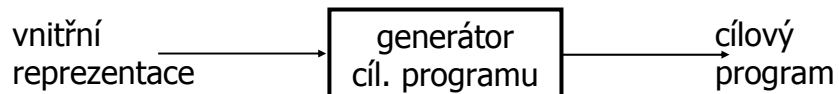
- překladač potřebuje informace o programu jako celku
  - obtížně proveditelné v modulárních programech
  - často jen v rámci jednoho podprogramu (intraprocedurální optimalizace)
- tyto informace se připojují k základním blokům

## Optimalizace – Příklady globálních optimalizací



- využití informací o *živosti proměnných* k přidělování registrů a ke kontrole správné inicializace proměnných
- odstranění redundantních výpočtů analýzou *společných podvýrazů*
- šíření konstant pomocí *dosahujících definic*
- nahrazení volání podprogramu jeho tělem (otevřené podprogramy – inline)

## Generování cílového kódu - Činnost generátoru



- vytvoření sémanticky ekvivalentního programu
- výběr 'optimální' varianty

## Generování cílového kódu - Cílový jazyk (1)



- Absolutní strojový kód
  - nejefektivnější výstup
  - program se musí přeložit celý najednou (složitá vazba na knihovny)
  - vhodné pro malé programy, výukové překladače, specializované procesory
- Přemístitelný strojový kód
  - běžné používání v klasických překladačích
  - nutnost použití sestavovacího programu

## Generování cílového kódu - Cílový jazyk (2)



- Jazyk symbolických instrukcí
  - jednoduché generování
    - adresové výpočty, makroinstrukce
  - vyžaduje další dva průchody assembleru
- Jazyk virtuálního procesoru
  - Java VM
  - .NET MSIL

## Generování cílového kódu - Požadavky na generátor cílového programu



- Rychlý překlad -> rychlé algoritmy
- Bezpečný kód -> kontroly v době běhu
- Dobrá detekce chyb za běhu
- Vlízký vztah mezi zdrojovým a cílovým programem -  
> jednodušší ladění
- **Efektivita cílového programu** (rychlost+velikost)
- *Protichůdné požadavky!*

## Generování cílového kódu - Řešení



- Sada překladačů - spíš historie
  - IBM FORTRAN - FORTRAN H
  - PL/I Checkout compiler - diagnostika  
Full compiler  
Optimizing compiler
- Parametrizovatelný překladač
  - Zapnutí/vypnutí optimalizací
  - Výběr prováděných optimalizací

## Generování cílového kódu - Klasické metody generování cílového programu



- Princip
  - dekompozice prvků jazyka na nezávislé podproblémy (např. součet dvou hodnot)
  - každý podproblém řešíme tou nejefektivnější metodou
- Implementace
  - sada procedur pro konkrétní podproblémy