Západočeská univerzita
FAKULTA APLIKOVANÝCH VĚD

Okruhy otázek ke státní závěrečné zkoušce z předmětu
Systémové programování (SP)

Operační systémy (OS)
Paralelní programování (PPR)
Formální jazyky a překladače (FJP)
Výkonnost a spolehlivost čísl. systémů (VSP)

Studijní program:	3902	Inženýrská informatika
Obor:	2612T025	Informatika a výpočetní technika – Softwarové inženýrství
	3902T031	Softwarové inženýrství

Akademický rok: 2005/2006

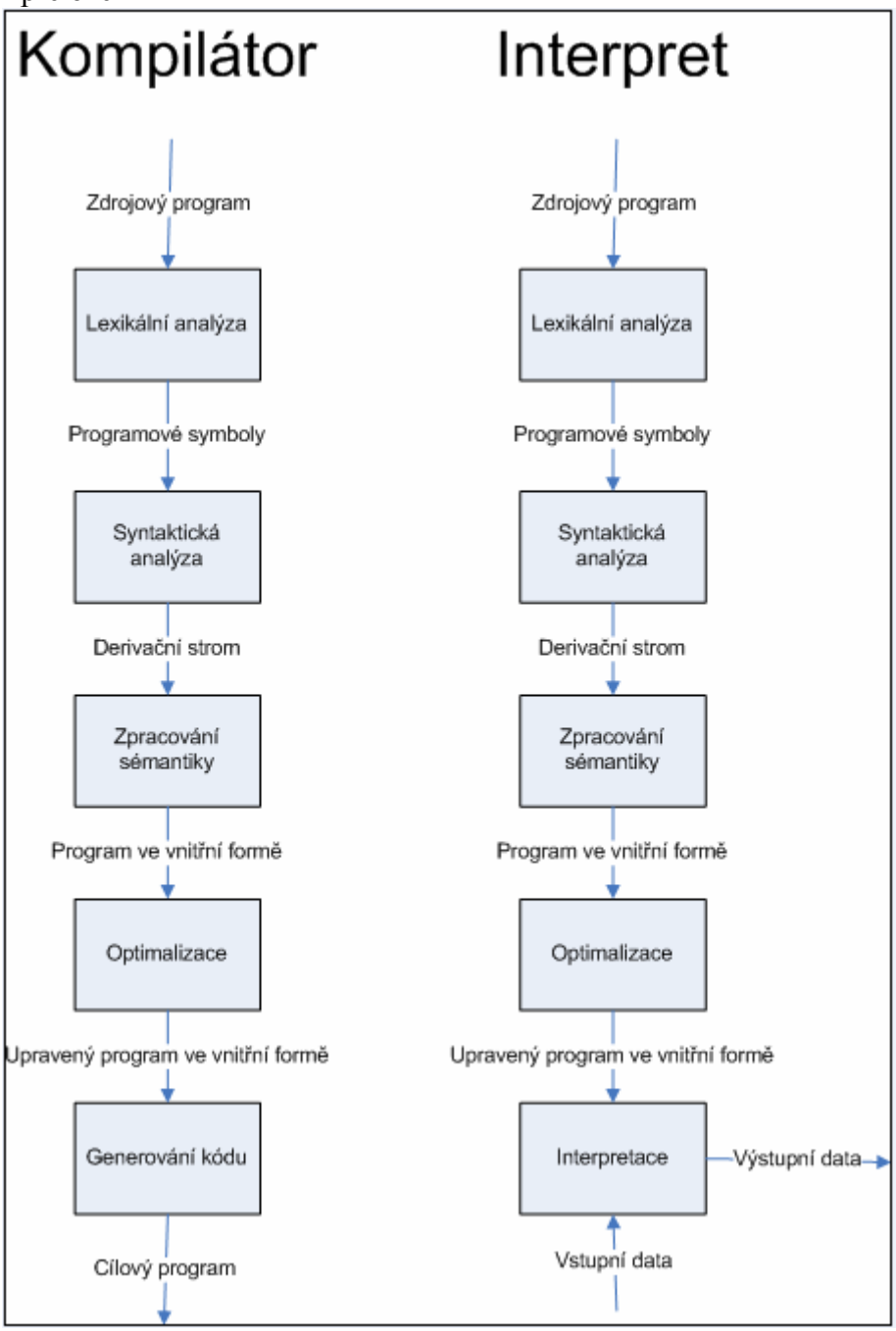
Obsah:

1 Struktura a princip činnosti překladače

Překladač – obvykle program, který čte zdrojový program a převádí ho do cílového jazyka, zdrojový program je napsaný ve zdrojovém jazyce, cílový program v cílovém jazyce, důležitou části překladače jsou diagnostické zprávy (informování o chybách ve zdrojovém programu)

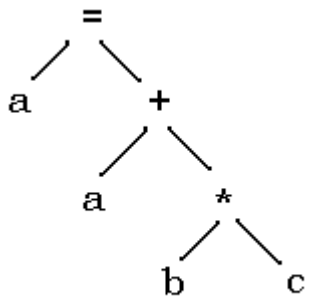
Dva typy překladačů:

Kompilátor – všechny příkazy překládá najednou, program lze spustit až po ukončení celého překladu (většina programovacích jazyků, Pascal, Java, C, …)
Interpret – zpracovává příkazy jednotlivě a každý provede okamžitě po jeho přeložení



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

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



Sémantická analýza – provádějí se některé kontroly, zajišťující správnost programu z hlediska vazeb, které nelze provádět v rámci syntaktické analýzy (např. kontrola deklarací, typová kontrola, apod.).

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

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

2 Regulární gramatika, konečný automat a jejich ekvivalence

Gramatika - Gramatika G je čtveřice (N,Σ,P,S), kde:

- N je konečná množina *neterminálních symbolů* (neterminálů).
- Σ je konečná množina *terminálních symbolů* tak, že žádný symbol nepatří do N a Σ zároveň.
- P je konečná množina *odvozovacích pravidel*. Každé pravidlo je tvaru

$$(\Sigma \cup N)^* \longrightarrow (\Sigma \cup N)^*$$

- S je prvek z N nazývaný *počáteční symbol*.

Regulární gramatika – je to gramatika typu 3 (podle Chomského hierarchie). Pravidla těchto gramatik jsou omezena na jeden neterminál na levé straně. Pravá strana se skládá z řetězce terminálů, který může být následován jedním neterminálem, tedy:

$$X \to wY$$
$$X \to w,$$

kde X, Y jsou neterminály a w je řetězcem terminálů. Regulární gramatiky se také nazývají **pravé lineární gramatiky**. Obdobně se definují i **levé lineární gramatiky**, které obsahují pravidla typu:

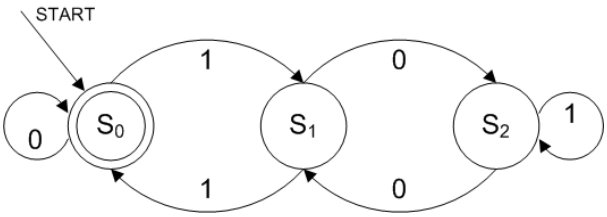
$$X \to Yw$$
$$X \to w$$

Pravé a levé gramatiky jsou ekvivalentní. Jazyky generované regulárními gramatikami jsou právě jazyky rozpoznatelné konečným automatem.

Konečný automat – formálně je konečný automat definován jako uspořádaná pětice (S,Σ,P,s,F), kde:

S je konečná množina stavů.
Σ je konečná množina vstupních symbolů nazývaná abeceda.
P je tzv. přechodová funkce (též přechodová tabulka), popisující pravidla přechodů mezi stavy.
s je počáteční stav ($s \in S$)
F je množina koncových stavů ($F \in S$)

Popis činnosti automatu:
Na počátku se automat nachází v definovaném počátečním stavu. Dále v každém kroku přečte jeden symbol ze vstupu a přejde do stavu, který je dán hodnotou, která v přechodové tabulce odpovídá aktuálnímu stavu a přečtenému symbolu. Poté pokračuje čtením dalšího symbolu ze vstupu, dalším přechodem podle přechodové tabulky, atd.
Podle toho, zda automat skončí po přečtení vstupu ve stavu, který patří do množiny koncových stavů, platí, že automat buď daný vstup *přijal* nebo *nepřijal*. Množina všech řetězců, který daný automat přijme, tvoří regulární jazyk.



Obrázek 1 - Znázornění konečného automatu

Postup převodu gramatiky na konečný automat

1. Potřebujeme získat gramatiku typu 3 ve standardní formě. Regulární gramatika je ve **standardní formě**, jestliže obsahuje pouze pravidla tvaru $X \to aY$ a $X \to \lambda$, kde *X,Y* jsou neterminály, *a* je právě jeden terminál. Toho dosáhneme takto:

   Původní gramatika typu 3: G = (N, T, S, P)
   Požadovaná regulární gramatika: G' = (N', T, S, P')

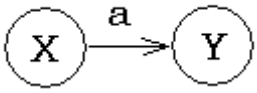
   Požadovaná gramatika G' bude mít stejné terminální symboly a stejný počáteční stav.

   Konstrukce přechodů P':
   a) do P' zařadíme všechna pravidla z P ve tvaru $X \to aY$ a $X \to e$, kde $X,Y \in N$ a $a \in T$.
   b) za každé pravidlo $X \to x_1 x_2 ... x_n Y$, kde $X,Y \in N$ a $x_i \in T$ zařadíme do P' soustavu pravidel:

   $X \to x_1 X_2$
   $X_2 \to x_2 X_3$
   ...
   $X_{n-1} \to x_{n-1} X_n$
   $X_n \to x_n Y$
   c) za každé pravidlo $X \to z_1 z_2 .. z_n$, kde $X \in N, z_i \in T$ zařadíme do P' soustavu:
   $X \to z_1 Z_1$
   $Z_1 \to z_2 Z_2$
   ...
   $Z_{n-1} \to z_n Z_n$
   $Z_n \to e$
   d) místo pravidel tvaru $X \to Y$, kde $X,Y \in N$ zařadíme do P' soustavu pravidel ve tvaru: $Z' \to zZ''$ pro $\forall Z' \in U(Y)$ a $\forall Y \to zZ'' \in P$, zároveň platí, že $U(Y) = \{X \mid X \Rightarrow Y\}$
   Jinými slovy, musíme zajistit to, aby z každého stavu X pro, který máme $X \to Y$, bylo možné odvodit všechny řetězce, které lze odvodit z Y.
   e) N' vznikne obohacením N o všechny nově vytvořené neterminální symboly

2. Zkonstruujeme automat z nově vytvořené gramatiky tímto formálním postupem:

   - stavy budou odpovídat neterminálním symbolům
   - vstupy budou odpovídat terminálním symbolům
   - přechodovou funkci zkonstruujeme na základě analogií

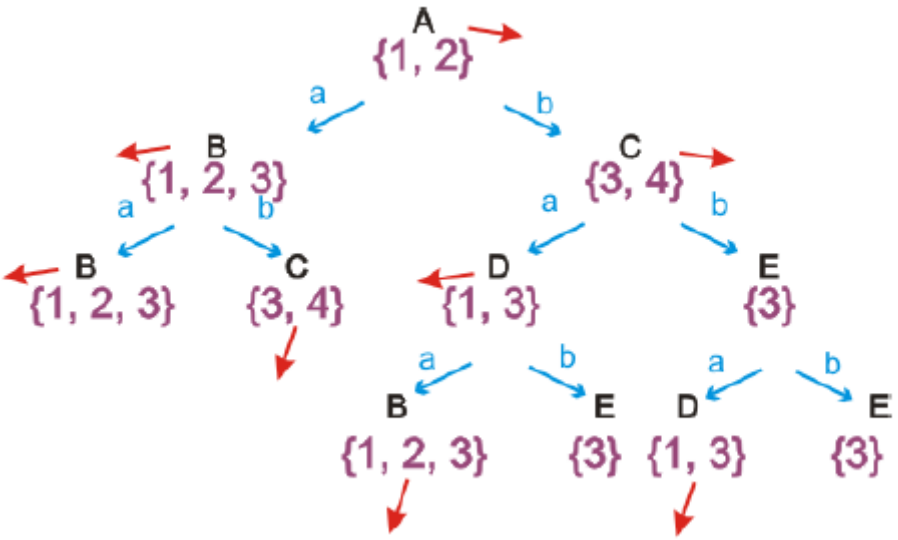
$X \to aY \in P \quad \Rightarrow$ 

   - počáteční stav bude odpovídat počátečnímu symbolu
   - množinu koncových stavů určíme tak, že:
     $X \to e \in P \quad \Rightarrow \quad X \in F$

Tímto jsme získali **nedeterministický automat**, kde můžeme nalézt 3 typy nedeterminismu:
- nejednoznačně určený počáteční stav
- nejednoznačné přechody (např. $X \to aY$ a zároveň $X \to aZ$)
- e přechody

Převod na deterministický automat:

Začínáme s množinou všech vstupních stavů a hledáme a zjišťujeme pro každý z terminálů do jakých všech stavů se z výchozího stavu můžeme dostat, doufám že následující obrázek bude dostatečně názorný.



3 Lexikální analýza

Úkolem lexikálního analyzátoru je nalezení a **rozpoznávání lexikálních symbolů.** Zdrojový program vstupuje do procesu překladu jako posloupnost znaků. Tato posloupnost se čte lineárně zleva doprava a sestavují se z ní lexikální symboly (tokens). Vynechávají se nevýznamné mezery, konce řádků a komentáře. Příkladem lexikálního elementu je `číslo`, `identifikátor`, klíčová slova (**`if, begin,`** …), =, := apod. Všechny rozpoznané lexikální symboly se ukládají do vnitřních struktur programu, tyto struktury jsou již snadno dále zpracovatelné. Vnitřní struktury většinou obsahují identifikaci o jaký typ lexikálního symbolu se jedná, např.: IDENTIFIKÁTOR, ČÍSLO, KLÍČOVÉ_SLOVO_IF, KLÍČOVÉ_SLOVO_BEGIN, … Identifikátory a proměnné u sebe často ukládají dodatečné hodnoty, například u lexikálního symbolu ČÍSLO by se ukládala hodnota, která se vyhodnotí v průběhu lexikální analýzy.

Příklad:
Na vstupu: A = 10
Nalezené lexikální symboly:

IDENTIFIKÁTOR – přidružený atribut A
SYMBOL_PŘIŘAZENÍ
ČÍSLO – přidružený atribut 10

K tomu, aby lexikální analyzátor dovedl rozpoznat ve vstupní posloupnosti znaků jednotlivé lexikální elementy, se používá **konečný automat.** Příslušná gramatika by mohla vypadat např. takto:

```
lexikální element = identifikátor | klíčové_slovo | číslo |speciální_symbol
speciální_symbol = '+' | '*' | '/' | '-' | '=' | '(' | ')' | '.'
identifikátor = písmeno { písmeno }
klíčové_slovo = identifikátor
písmeno = 'a' | 'b' | .... | 'A' | 'B' | .... | 'Z'
číslo = číslice { číslice }
číslice = '0' | '1' | .... | '9'
```

K možným nejednoznačnostem může dojít v případě, že jeden symbol je prefixem jiného symbolu. Využívá se **pravidla nejdelší shody.**

Pomůcka pro vytváření lexikálních analyzátorů: FLEX

FLEX (Fast LEXical analyzer generator)
Program LEX slouží k tvorbě jiných programů, které mají cosi udělat se vstupním (textovým) souborem za pomoci lexikální analýzy. Tím se myslí analýza struktur, které se dají zapsat lineárními gramatikami, konečnými automaty nebo regulárními výrazy. Typické použití LEXu je dvojí: vytvořený program pracuje samostatně, nebo slouží jako vstupní filtr pro jiný (syntaktický) analyzátor, např. bison či yacc.

Vstupem programu LEX je soubor (obvykle s koncovkou .l), který popisuje rozpoznávaná slova a akce, které se mají po jejich rozpoznání provést. Slova se popisují regulárními výrazy, akce v cílovém programovacím jazyku. Výstup LEXu je zdrojový kód hotového programu, který se potom musí běžným způsobem přeložit. Pokud tedy používáme variantu LEXu, která generuje výstup v jazyku C, musí být i akce zapsané v jazyku C.

Soubory .l mají následující strukturu:

```
deklarace, definice
%%
popis slov a akcí
%%
další funkce (zapsané v cílovém jazyku)
```

Jako příklad si uvedeme program, který ve vstupním souboru nahradí všechny identifikátory slovem "IDENTIFIKATOR"

```
%%
[a-zA-Z_][0-9a-zA-Z_]* printf("IDENTIFIKATOR");
%%
int main(void)
{
  yylex();
  return 0;
}
```

V popisu rozpoznávaných slov lze používat následující konstrukce:

x	znak "x"
[xy]	znak "x" nebo "y"
[x-z]	všechny znaky od "x" až k "z"
[^x]	jakýkoliv znak vyjma "x"
.	jakýkoliv znak, až na novou řádku
^x	znak "x", pokud se nachází na začátku řádky
x$	znak "x", pokud se nachází na konci řádky
x*	libovolný počet znaků "x"
x+	alespoň jeden znak "x"
x?	jeden nebo žádný znak "x"
x{m,n}	M až n výskytů znaku "x"
x|y	znak "x" nebo "y"
(x)	znak "x"
x/y	znak "x", je-li následován znakem "y"
{DEF}	doplnění definice z úvodní sekce
<y>x	znak "x", je-li splněna podmínka y

4 Bezkontextová gramatika a zásobníkový automat

Bezkontextové gramatiky (BKG) jsou gramatiky typu 2 podle Chomského hierarchie. Skládají se z pravidel $A \to \gamma$ kde A je neterminál a γ je řetězec terminálů a neterminálů. Pravidlo $S \to e$ je povoleno, pokud se S nevyskytuje na pravé straně žádného pravidla. Tyto jazyky jsou právě jazyky rozpoznatelné nějaký nedeterministickým zásobníkovým automatem.

Bezkontextovou gramatiku si lze představit jako $G = (N, T, P, S)$, kde:

- N je množina neterminálních symbolů
- T je množina terminálních symbolů
- $S \in N$ a je to počáteční symbol
- P je množina přepisovacích pravidel ve tvaru $A \to \gamma$, kde $A \in N$ a $\gamma \in N \cup T$

Příkladem může být jazyk $L = \{0^n 1^n\}$ pro $n \geq 0$, takovýto jazyk není rozpoznatelný konečným automatem, zásobníkovým ano.
Pro tento jazyk by platilo:

$N = \{S\}$

$T = \{0,1\}$

$P = \{S \to 0S1, S \to e\}$

$S = \{S\}$

Zásobníkový automat

Formálně je zásobníkový automat definován jako uspořádaná sedmice $(Q, T, G, \delta, q_0, z_0, F)$, kde:

- Q je konečná množina vnitřních stavů,
- T je konečná vstupní abeceda,
- G je konečná abeceda zásobníku,
- δ je tzv. přechodová funkce, popisující pravidla činnosti automatu (jeho program), je definováno jako zobrazení $QxT \cup \{e\})xG^*$ do QxG^*
- q_0 je počáteční stav,
- z_0 popisuje symboly uložené na počátku v zásobníku,
- F je množina *přijímajících stavů*, $F \subseteq Q$.

Je vidět, že zásobníkový automat se v podstatě skládá z konečného automatu, který má navíc k dispozici potenciálně nekonečné množství paměti ve formě zásobníku. Obsah tohoto zásobníku ovlivňuje činnost automatu tím, že vstupuje jako jeden z parametrů do přechodové funkce.

Popis činnosti automatu

Na počátku se automat nachází v definovaném počátečním stavu a zásobník obsahuje pouze počáteční symboly. Dále v každém kroku podle aktuálního stavu, symbolů na vrcholu

zásobníku a symbolu na vstupu provede přechod, při kterém může vyjmout ze zásobníku několik symbolů, vložit místo nich jiné a na vstupu přečíst další symbol. Toto se opakuje.

Po dokončení činnosti (po přečtení celého vstupu, pokud do té doby nedojde k chybě) je rozhodnuto, jestli automat vstupní řetězec přijal. K tomu mohou sloužit dvě kritéria:

- stav, ve kterém se na konci automat nachází, patří do množiny přijímajících stavů, nebo
- zásobník je na konci prázdný.

Obě definice jsou ekvivalentní, automaty na sebe lze vzájemně převádět (u druhé možnosti je možno z definice automatu zcela vypustit množinu přijímajících stavů).

Konfigurace automatu se dá popsat uspořádanou trojicí (q, w, alfa), kde q je vnitřní stav, w dosud nezpracovaná část vstupu a alfa obsah zásobníku. Na počátku práce je automat v konfiguraci (q_0, w, z_0).

Vztah bezkontextových gramatik a zásobníkových automatů
Pro danou BKG gramatiku W=(N, T, P, S) můžeme sestrojit zásobníkový automat P takový, že L(W)=L(P). Jsou dvě varianty:

a) Konstrukce zásobníkového automatu, který je modelem syntaktické analýzy **shora dolů**:

- Q = {q} (automat má jen jeden vnitřní stav),
- T je shodná s množinou terminálních symbolů rozpoznávané gramatiky,
- G = N+T, tj. v zásobníku se může vyskytnout jakýkoliv symbol rozpoznávané gramatiky,
- δ je dáno rozkladovou tabulkou,
- q_0 = q, počáteční stav automatu je q, neboť automat jiné stavy nemá,
- z_0 = S, tj. na počátku je v zásobníku startovací symbol gramatiky
- F = {}, což se interpretuje jako "automat akceptuje vyprázdněním zásobníku".

b) Analýza zdola nahoru je obecnější a vyžaduje trochu složitější automat:

- Q = {q, r}, stav q je "pracovní", stav r "akceptační",
- T je shodná s množinou terminálních symbolů rozpoznávané gramatiky,
- G je v nejjednodušším případě rovno N+T+{#}, tj. sjednocení symbolů gramatiky a speciálního symbolu "#"; deterministický automat může mít množinu G složitější
- d je dáno rozkladovou tabulkou,
- q_0 = q,
- z_0 = #,
- F = {r}.

Příklad rozkladová tabulka pro danou gramatiku

```
1.   S --> dSA    (1)
2.   S --> bAc    (2)
3.   A --> dA     (3)
4.   A --> c      (4)
```

M	b	c	d
S	2		1
A		4	3

(Bližší vysvětlení bude nejspíš v dalších otázkách)

5 Derivace, derivační strom, jednoznačnost gramatik

Gramatika - Gramatika G je čtveřice (N,Σ,P,S), kde:

- N je konečná množina *neterminálních symbolů* (neterminálů).
- Σ je konečná množina *terminálních symbolů* tak, že žádný symbol nepatří do N a Σ zároveň.
- P je konečná množina *odvozovacích pravidel*. Každé pravidlo je tvaru

$$(\Sigma \cup N)^* \longrightarrow (\Sigma \cup N)^*$$

- S je prvek z N nazývaný *počáteční symbol*.

Jazyk je vlastně množina slov, slova se skládají z písmen – tedy z terminálů. Gramatiky popisují jazyk.

P je množina pravidel, pomocí kterých lze odvodit jazyk J. $S \in N$ je výchozí neterminál. Základní operací při vyvozování slov jazyka J je tzv. substituce, kdy za neterminály dosazujeme pravidla, která je definují. Proces postupných substitucí se nazývá **derivace**. Cílem derivace je vyvodit z výchozího neterminálu S platné slovo jazyka J.

Derivační strom říká v jakém pořadí byla jednotlivá přepisovací pravidla aplikována, každé patro derivačního stromu odpovídá derivaci – tedy substituci neterminálního symbolu přepisovacím pravidlem.

Derivační strom je orientovaný acyklický graf, který má jediný kořen, do všech ostatních uzlů vstupuje právě jedna hrana, a dále má tyto vlastnosti:

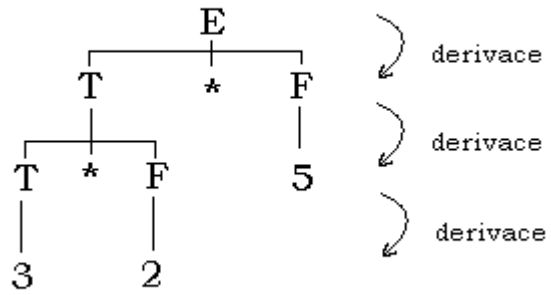
- **Kořen** stromu je ohodnocen **startovacím symbolem** gramatiky
- **Listy** jsou ohodnoceny **terminálními symboly**, všechny **ostatní** uzly jsou ohodnoceny **neterminálními symboly**
- Derivační strom tvoříme zleva doprava a shora dolů, proto není třeba značit orientaci hran

Příklad derivace:

Máme gramatiku G = (N,Σ,P,S) s přepisovacími pravidly:

E → E + T | T
T → T * F | F
F → (E) | i

derivační strom



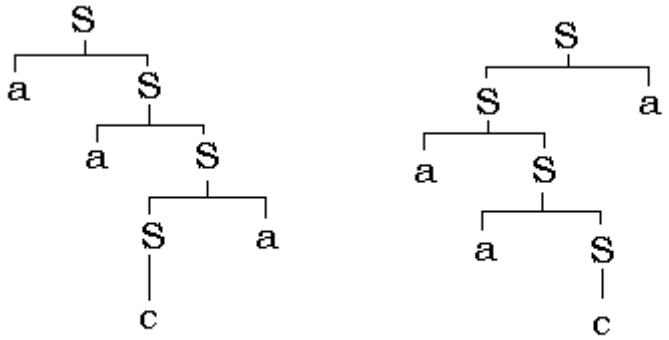
Víceznačnost gramatik

Věta generovaná gramatikou G je **víceznačná**, existují-li alespoň dva různé derivační stromy této věty. G pak rovněž nazýváme víceznačnou.

Nutnou podmínkou jednoznačnosti gramatiky je, aby pro žádný neterminální symbol neexistovalo jak pravidlo rekurzivní zprava, tak i pravidlo rekurzivní zleva.

Ukázka nejednoznačnosti pro gramatiku s přepisovacími pravidly (pro větu `aaca` existuje více než jeden derivační strom):

S → aS | Sa | c



Víceznačnost není dobrá vlastnost. Levorekurzivní gramatiku nelze použít k analýze shora dolů

Odstranění levé rekurze

Nechť je dána BKG G = (N, T, P, S), ve které,

A -> Aα1 | Aα2 | ... | Aαm | β1 | β2 | ... | βn

jsou všechna A pravidla v P a žádné, z β nezačíná A.

Pak G' = (N ∪ { A'}, T, P', S), kde P' obsahuje místo uvedených pravidel pravidla:

A -> β1 | β2 | … | βn | β1 A'| β2 A'| … | βa A'
A' -> α1 | α2 | … | αm | α1 A' | α2 A'| …| αm A'

Je ekvivalentní s gramatikou G

Alternativa:

A -> β1 A'| β2 A'| … | βa A'

A' -> α1 A' | α2 A'| …| αm A' | e

Př.:
E → E + T | T
=>
E → T | T E'
E' → +T | +T E'

Algoritmus pro odstranění nepřímé levé rekurze (to se doufam nemá cenu učit):

1. Zvolíme uspořádání na N = {A1, A2, ... An} tak, aby platilo:
   je-li Ai -> α pravidlo, jehož pravá strana začíná neterminálním symbolem Aj , pak j > i .
   Přiřaďme i := 1

2. Odstraníme přímou levou rekurzi u Ai pravidel (postup viz výše)

3. Je-li i = n pak jsme získali výslednou G' a skonči

   Jinak přiřaď i := i + 1 ; j := 1

4. Každé, pravidlo tvaru Ai -> Aj γ nahraď pravidly

   Ai -> α1 γ | α2 γ| … | αp γ, kde

   Aj -> α1 | α2 | … | αp jsou všechna Aj pravidla

5. Je-li j = i - 1 jdi na krok 2., jinak j := j + 1 a jdi na 4.

6 Metody syntaktické analýzy, rekurzivní sestup, principy LL analýzy

Při syntaktické analýze konstruujeme derivační strom. Podle toho, jak je konstruován derivační strom věty, rozlišujeme dvě základní metody syntaktické analýzy: u metody shora dolů derivační strom konstruujeme od kořene k listům a zleva doprava, u metody zdola nahoru postupujeme od listů ke kořeni, avšak také zleva doprava.

Jednotlivé metody budou ukázány na příkladu akceptace řetězce "abaaab" následující gramatikou:

```
S --> aAS      (1)
S --> b        (2)
A --> bA       (3)
A --> a        (4)
```

Metoda shora dolů
Derivační strom konstruujeme shora od kořene (ohodnoceného startovacím symbolem gramatiky) dolů k listům, zleva doprava, podle levé derivace. Jedná se o zásobníkový automat. Počáteční konfigurace automatu se dá popsat uspořádanou trojicí (q, w, alfa), kde q je vnitřní stav, w dosud nezpracovaná část vstupu a alfa obsah zásobníku. Na počátku práce je automat v konfiguraci (q_0, w, z_0), pro náš případ tedy (q, abaaab, S).
Pokud prostě generujeme větu v gramatice, můžeme v případě více pravidel se stejnou levou stranou náhodně vybírat. Naším úkolem však bývá analýza již existující věty (programu). Zde již náhoda nepřichází v úvahu, protože posloupnost pravidel pro levou derivaci již musí být jednoznačná. Potřebujeme automat, který tuto analýzu provádí, a tento automat musí mít možnost jednoznačně vybírat mezi pravidly to správné. Existují dva postupy:

- Analýza s návratem – postupně zkoušíme vhodná pravidla. Nejdřív první, pokračujeme dále ve výpočtu, a když se ukáže, že pravidlo nevyhovuje (dostaneme se do „slepé“ uličky), vrátíme se zpátky a vyzkoušíme druhé pravidlo, když to nevyhovuje, tak třetí, … Tato metoda je sice účinná, ale zbytečně pomalá, proto se moc nepoužívá.
- Deterministická analýza – při výběru pravidla se řídíme dalšími informacemi. Může to být „pohled do budoucnosti“, kdy se díváme dále do vstupní posloupnosti symbolů a řídíme se tím, co později dostaneme na vstupu, nebo například kontrola obsahu zásobníku (nestačí nám pouze vidět ten symbol, který ze zásobníku vyjímáme, ale i další, které jsou pod ním)

```
(q, abaaab, S) :-        start
(q, abaaab, aAS) :-      rozklad podle (1), dále jen (1)
(q, baaab, AS) :-        srovnání terminálních symbolů, dále jen
srovnání
(q, baaab, bAS) :-       (3), rozkládá se vždy nejlevější symbol
(q, aaab, AS) :-         srovnání
(q, aaab, aS) :-         (4)
(q, aab, S) :-           srovnání
(q, aab, aAS) :-         (1)
(q, ab, AS) :-           srovnání
(q, ab, aS) :-           (4)
(q, b, S) :-             srovnání
(q, b, b) :-             (2)
(q, e, e)                akceptováno
```

Metoda zdola nahoru
Při použití metody zdola nahoru konstruujeme derivační strom zdola od listů nahoru ke kořeni, také postupujeme zleva doprava, protože tímto směrem se obvykle čte text nebo třeba soubor.
Stejně jako u první metody, i zde budeme používat lineární rozklad, tentokrát pro pravou derivaci:
Pravý rozklad věty v gramatice G je obracená posloupnost čísel pravidel použitých v pravé derivaci této věty v gramatice G. Proč obracená posloupnost? My totiž při pravé derivaci v gramatice generujeme větu zprava doleva (přepisujeme vždy neterminál nejvíce vpravo), ale automat čte vstup zleva doprava, tedy tento postup obrací. Proto vytváří obrácenou posloupnost pravidel k té, kterou bychom použili při generování věty.
Syntaktická analýza metodou zdola nahoru je proces nalezení pravého rozkladu dané věty. Podobně jako u metody shora dolů, i zde se musíme rozhodovat mezi pravidly, která chceme použít. Tentokrát však nejde o pravidla se stejnou levou stranou (pro stejný neterminál), ale rozhodujeme se mezi pravidly, která mají podobnou pravou stranu a jsou proto použitelná pro tentýž podřetězec větné formy. Řešíme to podobně jako u předchozí metody:

- Analýza s návratem – vybereme ve větné formě jeden podřetězec (jako první vybíráme ten, který začíná nejvíc vlevo, je co nejdelší a je shodný s pravou stranou některého pravidla), přepíšeme neterminálem na pravé straně pravidla a pokračujeme v konstrukci derivačního stromu. Když zjistíme, že tento krok nevede k úspěchu, vyzkoušíme jiný podřetězec, . . . Tato metoda je jako v předchozím případě také časově náročná, proto není moc používaná.

- Deterministická analýza – využíváme další informace získané při překladu, například obsah nepřečtené části vstupní pásky nebo obsah zásobníku.

```
(q, abaaab, #) :-        start
(q, baaab, #a) :-        přijetí terminálního symbolu, dále jen přijetí
(q, aaab, #ab) :-        přijetí
(q, aab, #aba) :-        přijetí
(q, aab, #abA) :-        redukce podle pravidla (4), dále jen (4);
redukují se nejpravější symboly
(q, aab, #aA) :-         (3)
(q, ab, #aAa) :-         přijetí
(q, b, #aAaa) :-         přijetí
(q, b, #aAaA) :-         (4)
(q, e, #aAaAb) :-        přijetí
(q, e, #aAaAS) :-        (2)
(q, e, #aAS) :-          (1)
(q, e, #S) :-            (1)
(r, e, e)                akceptováno
```

Třídy jazyků LL(k)

Zkratka LL(k) znamená:
• Left to Right – vstupní text (soubor) čteme zleva doprava,
• Left Parse – vytváříme levý rozklad,
• při rozhodování mezi pravidly potřebujeme vidět nejvýše k znaků z nepřečtené části vstupu.

Gramatika je **typu LL(k)**, jestliže ji lze použit pro deterministickou syntaktickou analýzu metodou shora dolů (vytváříme levý rozklad) a při rozhodování mezi pravidly potřebujeme znát nejvýše k symbolů ze vstupu.

Jazyk je **typu LL(k),** pokud je generován některou LL(k) gramatikou.

Nechť G je bezkontextová gramatika. G je **silná LL(k) gramatika**, jestliže pro jakákoliv dvě pravidla se stejnou levou stranou $A \rightarrow \alpha$, $A \rightarrow \beta$, kde $\alpha \neq \beta$, platí
$FIRST_k(\alpha \circ FOLLOW_k(A)) \cap FIRST_k(\beta \circ FOLLOW_k(A)) =$ prázdná množina

Gramatiku, která je LL(k), ale není silná LL(k), nazýváme **slabá LL(k) gramatika**.

7 Vnitřní jazyky překladače

Po ukončení syntaktické a sémantické analýzy generují některé překladače explicitní *intermediární* reprezentaci zdrojového programu (**mezikód**). Intermediární[1] reprezentaci můžeme považovat za program pro nějaký abstraktní počítač. Tato reprezentace by měla mít dvě důležité vlastnosti: měla by být **jednoduchá pro vytváření** a **jednoduchá pro překlad do tvaru cílového programu.**
Intermediární kód slouží obvykle jako **podklad pro optimalizaci a generování cílového kódu**. Může však být také konečným produktem překladu v interpretačním překladači, který vygenerovaný mezikód přímo provádí.
Intermediární reprezentace mohou mít **různé formy.**

Postfixová notace
- operátory následují ihned za operandy
- A B C * D + - => A – (B * C + D)
- zpracování pomocí zásobníku, musíme vědět prioritu operátorů

Prefixová notace
- operátory a pak operandy
- * + A B + C D => (A + B) * (C + D)

Tříadresový kód

Abstraktní forma mezikódu sestávající ze sekvence příkazů ve tvaru **x := y op z**, kde *x, y* a *z* jsou jména, konstanty nebo dočasné proměnné, *op* je nějaký operátor.

Překlad výrazu *x+y*z* na tříadresový kód:

*t1 := y * z*
t2 := x + t1

Trojice a čtveřice

Implementací tříadresového kódu jsou záznamy se třemi nebo čtyřmi poli: trojice resp. čtveřice. Následující příklady budou ukázány na výrazu: a := b * (-c) + d [b]

Čtveřice

Záznam má čtyři položky nazývané *op, arg1, arg2* a *res*. Tříadresový příkaz ve tvaru *x := y op z* je reprezentován umístěním *op* do *op, y* do *arg1, z* do *arg2* a *x* do *res*. Některé tříadresové příkazy nepotřebují všechny položky (např. *x := y*).

Trojice

Jestliže se chceme vyhnout generování dočasných proměnných, je možné použít formu trojic. Trojice obsahuje *op, arg1* a *arg2*. Místo dočasných proměnných jsou indexy do pole trojic.

[1] Intermediární = jsoucí mezi dvěma jevy, přechodový, zprostředkující

Příklady

Ukažme si tříadresový kód, čtveřice a trojice na příkladě výrazu:

a := b * (- c) + d [b]

Tříadresový kód

```
t1 := - c
t2 := b * t1
t3 := d [ b ]
t4 := t2 + t3
a := t4
```

Čtveřice

	op	arg1	arg2	res
(1)	uminus	c		t1
(2)	*	b	t1	t2
(3)	loadidx	d	b	t3
(4)	+	t2	t3	t4
(5)	:=	t4		a

Trojice

	op	arg1	arg2
(1)	uminus	c	
(2)	*	b	(1)
(3)	loadidx	d	b
(4)	+	(2)	(3)
(5)	:=	a	(4)

8 Tabulka symbolů

Jakmile syntaktický analyzátor najde určitou konstrukci symbolů, tedy frázi, je třeba této konstrukci přiřadit význam.
Součástí syntaktického analyzátoru bývá procedura (nebo více procedur či funkci), která je postupně pro každou frázi volaná a jejím úkolem je doplnit údaje do tabulky symbolů nebo do interního kódu.

Do tabulky symbolů (tabulky objektů) **ukládáme** postupně všechny objekty - pojmenované **identifikátory** (které nejsou klíčovými slovy), **proměnné** nebo **konstanty**, **uživatelské datové typy**, **funkce**, **procedury**, **návěští** apod., na které v kódu narazíme. Pojem objekt zde budeme chápat obecněji než je obvyklé v teorii programování, bude to prostě jakýkoliv identifikátor, který není klíčovým slovem a lexikální analýza ho proto ještě neodlišila od jiných identifikátorů.

Zapisujeme zde obvykle **název**, **typ**, **adresu, případně počáteční hodnotu objektu**, **počet a typ parametrů funkce** a další informace potřebné při dalším překladu, ale také při provádění programu.

Tabulka symbolů může vypadat takto:

Název	Typ	Délka	Deklarováno	Adresa	Použito
`delky`	integer array 10	40 B	A	.....	N
`I`	byte	1 B	A	.....	A
`pocet`	integer	4 B	A	.....	N
`x1`	real	6 B	A	.....	N
`z1`	*nedefinováno*	0	N	0	A

V tabulce vidíme objekty délky (pole o délce 10 prvků, prvky jsou celá čísla), *I*, *pocet* a *x1*, které již byly deklarovány a objekt *I* také použit. Objekt *z1* ještě nebyl deklarován, ale už je v kódu použit. V jazyce, který umožňuje pracovat pouze s deklarovanými proměnnými, se jedná o sémantickou chybu.

U každého typu objektu potřebujeme uchovávat různé druhy informací. Například u proměnné je to název, adresa, datový typ, velikost potřebné paměti apod., u funkce název, adresa, návratový typ, počet a typ jednotlivých parametrů, příp. zda jsou volány hodnotou nebo odkazem (jestliže jsou volány odkazem, musí sémantický analyzátor navíc ošetřit, aby ve volání funkce byly jako skutečné parametry použity pouze názvy proměnných a nikoli například výrazy nebo konstantní hodnoty), u dalších typů objektů to budou opět jiné údaje. Řádky tabulky mohou být navzájem závislé (jeden uživatelský datový typ může využívat deklaraci již dříve uvedeného, popř. proměnná je typu deklarovaného dříve, . . .), nesmí se však jednat o kruhovou závislost.

Tato tabulka nám **slouží k mnoha účelům**. **Využívá ji zejména sémantický analyzátor** (kontroluje, zda proměnná použitá v kódu je deklarovaná a zda její datový typ odpovídá jejímu použití, jestli u funkce souhlasí počet a typ argumentů, atd.), používá se **také u generování cílového kódu** (překladač musí vědět, kolik místa v paměti má vyhradit pro jednotlivé symboly).

Při interpretaci obvykle není nutné uchovávat informaci o adrese, samotná tabulka symbolů může sloužit jako úschovna symbolů, se kterou pak neustále pracujeme.

Tabulka symbolů může být vytvářena již lexikálním analyzátorem, ten však má omezené možnosti při zjišťování některých údajů, proto je v mnoha případech vhodnější přenechat tuto práci syntaktickému nebo sémantickému analyzátoru. Často používaný postup je vytváření tabulky lexikálním analyzátorem (kdykoliv narazí na identifikátor, který není klíčovým slovem, uloží ho do tabulky) s tím, že další části překladače doplňují zbývající informace o vlastnostech uloženého identifikátoru.

Otázkou je, jak vlastně řadit jednotlivé objekty v tabulce. Důležitým kritériem je **rychlost vyhledávání**, protože k tabulce symbolů přistupuje zejména sémantický analyzátor velmi často. U jednodušších jazyků je možné tabulku automaticky řadit podle abecedy, u složitějších jazyků řešíme indexací, kdy zároveň s tabulkou vytváříme indexový seznam (příp. soubor), ve kterém jsou odkazy na objekty seřazené podle abecedy.

Speciální implementaci vyžaduje tabulka symbolů pro jazyk s blokovou strukturou, jako je třeba Pascal. Rozlišují se zde **lokální a globální objekty** a přístupnost lokálních je omezena. Každá proměnná je viditelná v tom bloku, ve kterém je deklarovaná, a také ve všech blocích vnořených.
Když v určitém bloku použijeme proměnnou, hledáme informace o ní nejdřív v tom bloku, ve kterém se nacházíme. Při neúspěchu se posouváme do nadřízeného bloku a tak postupujeme, dokud ji nenajdeme. Pokud neuspějeme ani v hlavním bloku, znamená to, že byla použita proměnná, která není deklarovaná, jde o sémantickou chybu. **Každý blok má svoji vlastní tabulku**.
S celou strukturou se pracuje jako s klasickým zásobníkem. Každá z tabulek má svou vlastní organizaci a je z ní přístupná nadřízená tabulka. „Aktivní tabulka je na vrcholu zásobníku, kde také začínáme prohledávat. **Při vyhodnocení konce bloku se aktivní tabulka ze zásobníku odstraní**. Po jejím odstraněni se sem přesune nadřízená tabulka. Tabulka hlavního bloku zůstává v zásobníku až do konce vyhodnocování programu, je odstraněna až jako poslední po vyhodnocení celého programu.

Implementace tabulky symbolů

- Vyhledávací netříděné tabulky (jen pro krátké programy)
  -prostá struktura
  -lineární seznam
- Vyhledávací setříděné tabulky
  -průběžné setřiďování
  -setřídění po zaplnění
- Frekvenčně uspořádané tabulky
- Binární vyhledávací stromy
- Tabulky s rozptýlenými položkami

Ukládání polí a struktur
Pole i struktury mají pevnou adresu začátku pole a pro přístup k jednotlivým prvkům se výsledná adresa dopočítává. Pole mohou být v paměti uložena buď po řádcích nebo po sloupcích. Tomu musí odpovídat **mapovací funkce**, která vypočítává relativní adresu prvků. K této adrese musí být připočtena adresa začátku pole.

9 Překlad jednoduchých jazykových konstrukcí

Překladač – obvykle program, který čte zdrojový program a převádí ho do cílového jazyka, zdrojový program je napsaný ve zdrojovém jazyce, cílový program v cílovém jazyce. Cílovým jazykem bývá většinou nějaký jazyk assemblerovského typu.

Příklad jednoduchých jazykových konstrukcí:

- **if** (větvení, různé podmínky – větší, menší, rovno, …, možnost else větve)
- **while** (podmíněný cyklus, podmínka na začátku, nemusí proběhnout ani jednou)
- **volání procedur**

Pro ukázku jak se překládají jednotlivé konstrukce na cílové instrukce je potřeba znát nějaký cílový jazyk – ve škole jsme probírali **PL/0** [pé el nula] ☺.

Překladač PL/0 tvoří cílový kód, který se skládá z následujících instrukcí:

```
lit 0,A          ulož konstantu A do zásobníku
opr 0,A          proveď instrukci A
         0       return
         1       unární minus
         2       +
         3       -
         4       *
         5       div
         6       odd
         8       =
         9       <>
        10       <
        11       >=
        12       >
        13       <=
lod L,A          ulož hodnotu proměnné z adr. L,A na vrchol zásobníku
sto L,A          zapiš do proměnné z adr. L,A hodnotu z vrcholu zásobníku
cal L,A          volej proceduru A z úrovně L
int 0,A          zvyš obsah top-registru zásobníku o hodnotu A
jmp 0,A          proveď skok na adresu A
jpc 0,A          proveď podmíněný skok na adresu A
```

Přeložení větvení pomocí if, konstrukce v ukázce se nachází uvnitř procedury, *i* a *j* jsou tedy lokálními proměnnými (v PL/0 odpovídají druhé úrovni vnoření– odkazování přes adresy 2,4 a 2,5)**:**

if (i > j) {
…
} else {
…
}

…
(10) TA 2,4 *uložení adresy do zásobníku (adresa proměnné i)*
(11) DR *dereference – ze zásobníku se vybere adresa, ze které se přečte číslo, které se uloží do zásobníku (hodnota i)*

(12) TA 2,5 *uložení adresy do zásobníku (adresa proměnné j)*
(13) DR *dereference – ze zásobníku se vybere adresa, ze které se přečte číslo, které se uloží do zásobníku (hodnota j)*
(14) REL GT *porovnání za i je větší než j (greater than)*
(15) IFJ 24 *pokud tato podmínka není splněna – skoč na kód else větve*
… zde by následoval kód, který se provede pokud je podmínka splněna …
(23) JU 31 (přeskočení else větve)
(24)… zde by následoval kód, který se provede pokud není podmínka splněna …
…
(31) zde pokračuje program (první příkaz po celém if bloku)

Přeložení cyklu while:

while (i != j) {
…
}

(4) TA 2,4 *adresa proměnné i*
(5) DR *dereference proměnné i*
(6) TA 2,5 *adresa proměnné j*
(7) DR *dereference proměnné j (obě jsou teď v zásobníku)*
(8) REL NE *jejich porovnání (nerovnost, not equal)*
(9) IFJ 32 *pokud nesplněno (tedy pokud platí i = j, pak skoč za cyklus, ukončení)*
...**zde by následovalo tělo cyklu …**
(31) JU 4 *prostý skok zpět na porovnání, jestli je podmínka již splněna*
(32) … zde pokračuje program (první příkaz po celém while bloku)

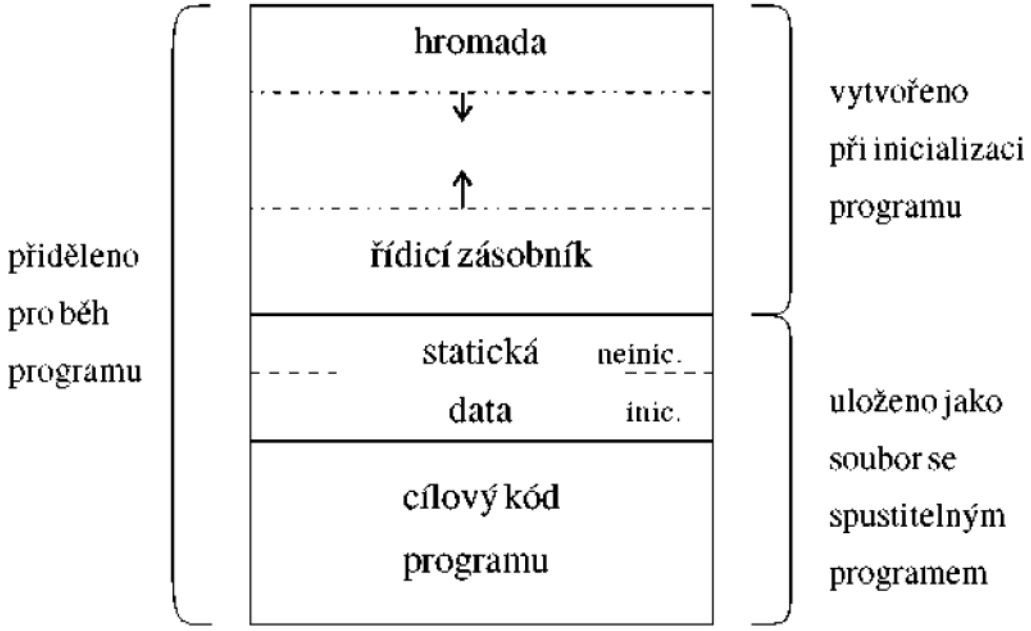
Vytvoření procedury:
(0) JU 37 skok na hlavní program (přeskočit kód procedury)
(1) BBEG 2,5 *začátek deklarace procedury, parametry jsou úroveň vnoření a velikost potřebné paměti*
(2) FPAR CONST *sem se uloží první předávaný parametr (přes zásobník)*
(3) FPAR CONST *sem se uloží druhý předávaný parametr*
… tělo procedury …
(36) RET *návrat z procedury*
… tady by byl hlavní program …
(52) CSUB 1 *ukázka volání procedury .. parametry se zadají v následujícím kroku*
(53) PAR 1,4
(54) PAR 1,5
… pokračování …

10 Statický a dynamický způsob přidělování paměti

Přeložený program dostane od operačního počítače k dispozici blok paměti, který obecně může být rozdělen na následující části:

- Vygenerovaný cílový kód
- Statická data
- Řídící zásobník
- Hromada

Velikost vygenerovaného **kód**u je známa již v době překladu, takže jej může překladač umístit **do staticky definované oblasti**, obvykle na začátek přiděleného paměťového prostoru. Rovněž velikost statických datových objektů může být známa již v době překladu a překladač je může umístit za program nebo uložit dokonce jako součást programu (to lze pouze u těch programovacích jazyků, které neumožňují rekurzivní volání procedur – Fortran). Jazyky umožňující rekurzi (Pascal, C, …) využívají **pro aktivace podprogramů řídicího zásobníku**, do kterého se ukládají jednotlivé aktivační záznamy.
Pro účely **dynamického přidělování** paměti (explicitně vyžadovaného voláním příslušných funkcí nebo implicitně při přidělování paměti například pro pole s dynamickými rozměry) se používá zvláštní část paměti zvané **hromada**. Vzhledem k tomu, že se velikosti použité části paměti pro zásobník a hromadu v průběhu činnosti programu mohou značně měnit, je výhodné pro obě části využít opačné konce společné části paměti – viz obrázek. **Nedostatek paměti se rozpozná tehdy, jestliže ukazatel konce některé oblasti překročí hodnotu ukazatele konce druhé oblasti**.



Pro zmíněné datové oblasti se používají následující hlavní **metody přidělování paměti**:

- Statické přidělování paměti v době překladu
- Přidělování paměti na zásobníku
- Přidělování paměti z hromady

Statické přidělování paměti v době překladu
Při statickém přidělování paměti jsou všem objektům v programu přiděleny **adresy již v době překladu**. Při kterémkoliv volání podprogramu jsou jeho lokální proměnné **vždy na stejném místě**, což **umožňuje zachovávat hodnoty lokálních proměnných nezměněné mezi různými aktivacemi podprogramu**. Statická alokace proměnných však klade na zdrojový jazyk určitá **omezení**. Údaje o velikosti a počtu všech datových objektů musejí být známy již v době překladu, **rekurzivní podprogramy** mají **velmi omezené** možnosti, neboť všechny aktivace podprogramu sdílejí tytéž proměnné, a konečně **nelze vytvářet dynamické datové struktury.**

Přidělování na zásobníku
Přidělování paměti pro aktivační záznamy na zásobníku se používá běžně u jazyků, které **umožňují rekurzivní volání podprogramů** nebo které používají staticky do sebe zanořené podprogramy. **Paměť pro lokální proměnné je přidělena při aktivaci podprogramu vždy na vrcholu zásobníku a při návratu je opět uvolněna**. To ale zároveň znamená, že **hodnoty** lokálních proměnných **se mezi dvěma aktivacemi podprogramu nezachovávají.**
Při implementaci přidělování paměti na zásobníku bývá jeden registr vyhrazen jako ukazatel na začátek aktivačního záznamu na vrchol zásobníku. Vzhledem k tomuto registru se pak počítají všechny adresy datových objektů, které jsou umístěny v aktivačním záznamu. Naplnění registru a přidělení nového aktivačního záznamu je součástí volací posloupnosti, obnovení stavu před voláním se provádí během návratové posloupnosti. Volací (a návratové) posloupnosti se od sebe v různých implementacích liší. Jejich činnost bývá rozdělena mezi volající a volaný program. Obvykle volající program určí adresu začátku nového aktivačního záznamu (k tomu potřebuje znát velikost záznamu vlastního), přesune do něj předávané argumenty a spustí volaný podprogram zároveň s uložením návratové adresy do určitého registru nebo na známé místo v paměti. Volaný podprogram nejprve uschová do svého aktivačního záznamu stavovou informaci (obsahy registrů, stavové slovo procesoru, návratovou adresu), inicializuje svá lokální data a pokračuje zpracováním svého těla. Při návratu opět volaný podprogram uloží hodnotu výsledku do registru nebo do paměti, obnoví uschovanou stavovou informaci a provede návrat do volajícího programu. Ten si převezme návratovou hodnotu a tím je volání podprogramu ukončeno.

Přidělování z hromady
Strategie přidělování na zásobníku je nepoužitelná, pokud mohou hodnoty lokálních proměnných přetrvávat i po ukončení aktivace, případně pokud aktivace volaného podprogramu může přežít aktivaci volajícího. V těchto případech přidělování a uvolňování aktivačních záznamů se mohou překrývat, takže nemůžeme paměť organizovat jako zásobník. Aktivační záznamy se mohou v těchto nejobecnějších situacích přidělovat z volné oblasti paměti (hromady), která se jinak používá pro dynamické datové struktury vytvářené uživatelem. Přidělené aktivační záznamy se uvolňují až tehdy, pokud se ukončí aktivace příslušného podprogramu nebo pokud už nejsou lokální data potřebná.
Při použití této strategie se pro vlastní přidělování a uvolňování paměti používají stejné techniky jako pro dynamické proměnné.

11 Principy interpretace a generování

Překladač je program, který k libovolnému programu PZ v jazyku JZ vytvoří program PC v jazyku JC se stejným významem.

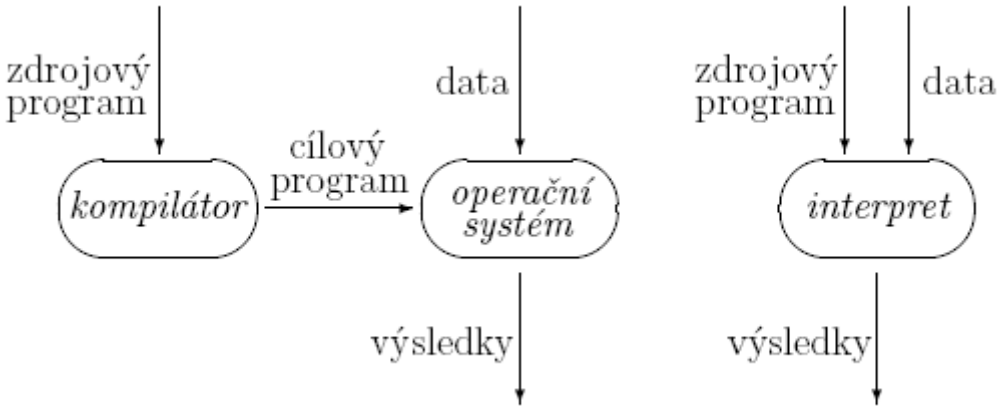
Nazýváme:
- PZ . . . zdrojový (překládaný) program,
- PC . . . cílový (přeloženy) program.

Překladač tedy zpracovává text zdrojového jazyka (zdrojový program) a převádí ho na sémanticky ekvivalentní text cílového jazyka (cílový program).

Překladače rozlišujeme podle typu cílového programu na dva druhy:

- **kompilátor** (generační překladač) je překladač, který má na vstupu program ve vyšším programovacím jazyce (Fortran, Pascal, C, C++, Delphi, . . .) a cílovým jazykem je strojový jazyk nebo jazyk symbolických instrukci (JSI, Assembler)

- **interpret** (interpretační překladač, někdy také interpreter) pouze interpretuje (provádí) zdrojový program pro zadaná vstupní data, tedy netvoří generovaný program, vytváří jen vnitřní reprezentaci programu pro svou vlastní potřebu (tu lze chápat jako cílový jazyk).



Každý z těchto druhů je vhodný pro jinou situaci. Zatímco cílový program kompilátoru (obvykle soubor obsahující strojový kód, např. EXE pro Windows) se provádí relativně velmi svižně, interpretovaný program je pomalý a pro mnoho vyšších programovacích jazyků proto nevhodný, protože překlad je prováděn při každém spuštění programu. U kompilátorů samotný překlad probíhá jen jednou, nesouvisí se samotným prováděním programu, což umožňuje provádět i časově náročné optimalizace a kontroly (logicky překlad u interpretace nesmí trvat moc dlouho, uživatel by si netrpělivostí „ukousal nehty“). Dalšími nevýhodami interpretačního překladače jsou jeho nezbytnost při spuštění interpretovaného programu a náročnost na paměťový prostor (při běhu musí být v paměti nejen zdrojový program, ale také celý překladač).
Ale i interpretační překladač má své výhody, může například umožňovat provedení pouze malé části zdrojového programu (např. u programovacího jazyka SmallTalk), jeho vytvoření

je jednodušší, programátor (autor překladače) obvykle nemusí ovládat assembler ani strojový jazyk a při výskytu chyby můžeme spolehlivěji určit její umístění.
Navíc v případě zdrojového programu, který vlastně uchováváme, jde obvykle o textový soubor, který obvykle zabírá mnohem méně místa než obdobný cílový program přeložený kompilátorem, a navíc je zdrojový program snadněji přenositelný (nejen proto, že se lépe „vměstná“ na jakékoliv paměťové médium, ale také je spustitelný prakticky na jakékoliv platformě – můžeme mít na strojích s různými operačními systémy nainstalován interpretační překladač pro tentýž jazyk, všechny tyto překladače přijmou tentýž zdrojový program).

Vlastnost	Kompilátor	Interpret
Rychlost běhu cílového programu	*lepší*	
Rychlost spuštění cílového programu	*lepší*[2]	
Rychlost překladu		*lepší*
Spotřeba paměti – operační (při běhu)	*lepší*	
Spotřeba paměti – cílový soubor na paměťovém médiu		*lepší*
Přenositelnost kódu mezi platformami (Win, Linux, Mac, ...)		*lepší*
Možnosti optimalizace	*lepší*	
Nezávislost na překladači	*lepší*	

Obrázek 2 - Srovnání kompilátorů a intepreterů

Interpretační jsou překladače starších verzí BASICu, shelly operačních systémů (např. Příkazový řádek Windows nebo shelly Unixových systémů), různé skriptovací jazyky pro jednotlivé operační systémy nebo Internet (např. Java Script) HTML, a také překladače čistě objektových jazyků (např. SmallTalk), kde se v průběhu provádění programu mohou dynamicky měnit typy objektů.
V praxi však překladače jednoho typu často využívají také výhod vlastností druhého typu. Interprety obvykle nejdřív program převedou („předkompilují“) do některého interního kódu vlastně jde o intermediální kód) a ten potom interpretují, částečně se tím zrychlí provádění programu, jen těsně po spuštěni programu je krátká časová prodleva. Kompilátory zase v sobě obvykle zahrnují možnost krokování, tedy provádění programu postupně po jednotlivých příkazech nebo blocích příkazů, což se dá chápat jako forma interpretace.