Překladače – typy, struktura a princip činnosti

Wednesday, May 29, 2013 4:50 PM

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

Typy

Postup při tvorbě cílového spustitelného programu:

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

Preprocesory

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

Kompilátory

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

Interprety

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

Java – kombinace kompilace a interpretace

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

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

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

Assemblery

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

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

Linker a loader

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

Typy překladačů

Formátory textu

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

Silikonový překladač

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

Dávkový překladač

Dávkové zpracování

Inkrementální překladač
Je interaktivní a překládá po úsecích

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

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

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

Paralelizující překladač
Zjišťuje nezávislost úseků programu

Optimalizující překladač
Možnosti ovlivnění optimalizace času či paměti programátorem.

Konverzační překladač - interaktivní

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

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

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

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

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

Výhodou kompilátoru je rychlá exekuce programu.

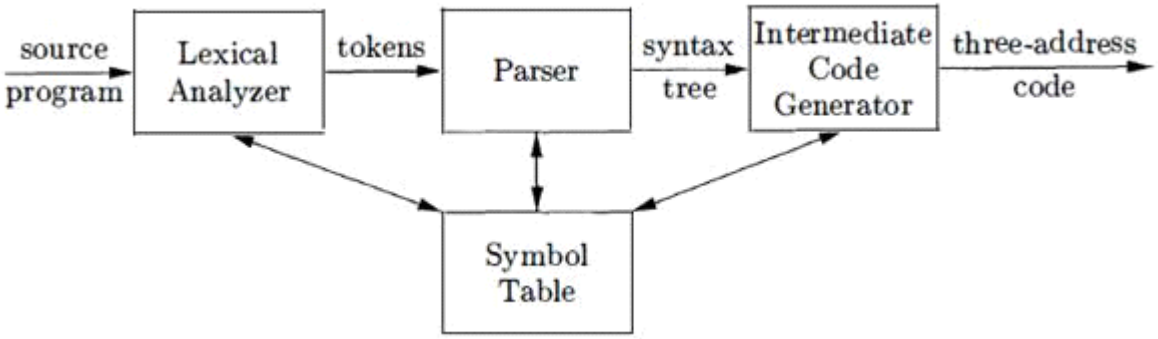
Struktura Interpreta
zpracovává příkazy jednotlivě a každý provede okamžitě po jeho přeložení (Python, Perl, JavaScript, Ruby, …)

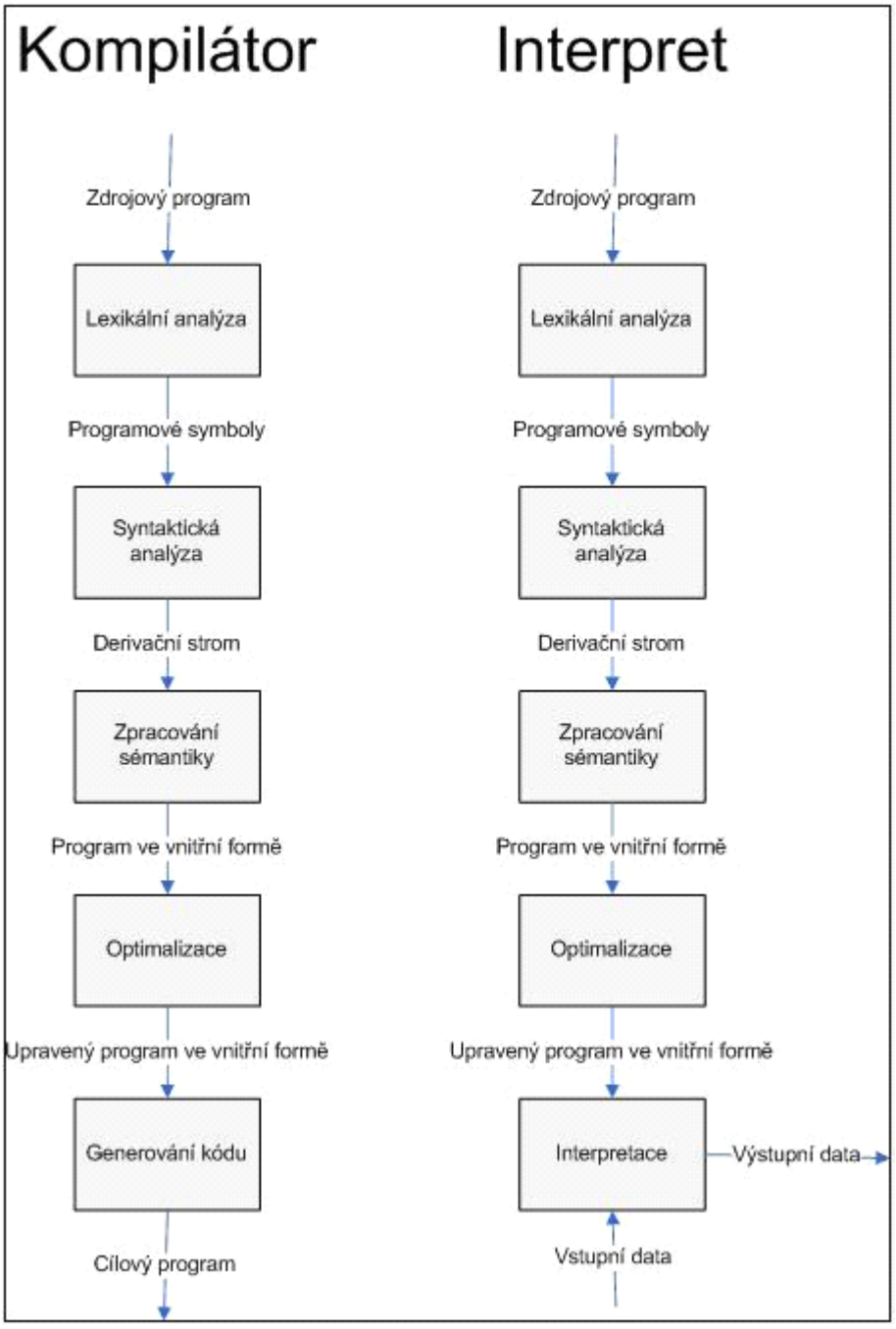
ANALÝZA: zdrojový program → **lexikální analýza** (lineární), programové symboly → **syntaktická analýza** (hierarchická), derivační strom

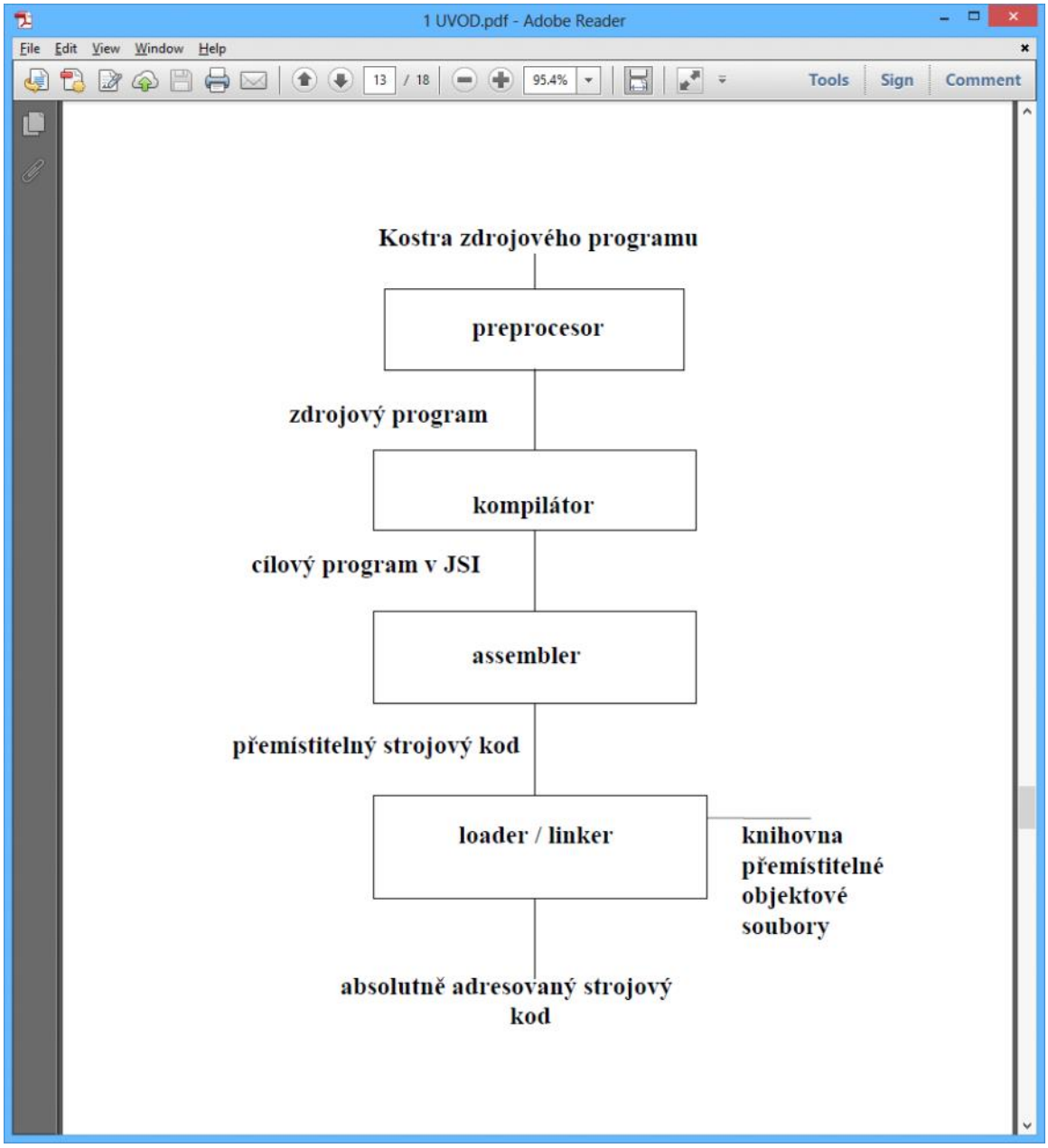
SYNTÉZA: derivační strom → **zpracování sémantiky**, program ve vnitřní formě → **optimalizace** (příprava generování), upravený program ve vnitřní formě → **interpretace**, pracuje se vstupními daty, aby vygeneroval výstupní data

Výhodou interpretu je:

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


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


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

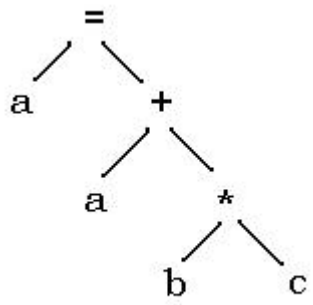


Lexikální analýza

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

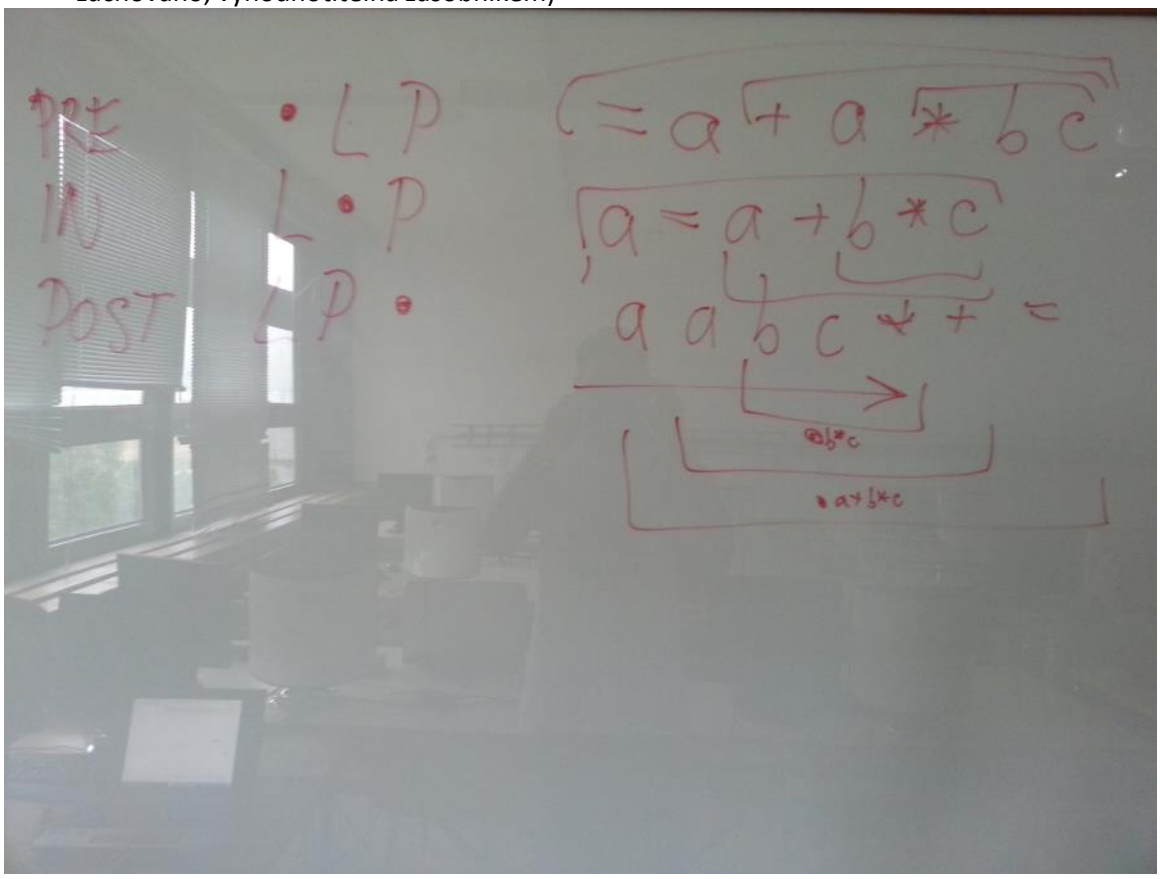
Syntaktická analýza

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



Notace vnitřního jazyka překladače:

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



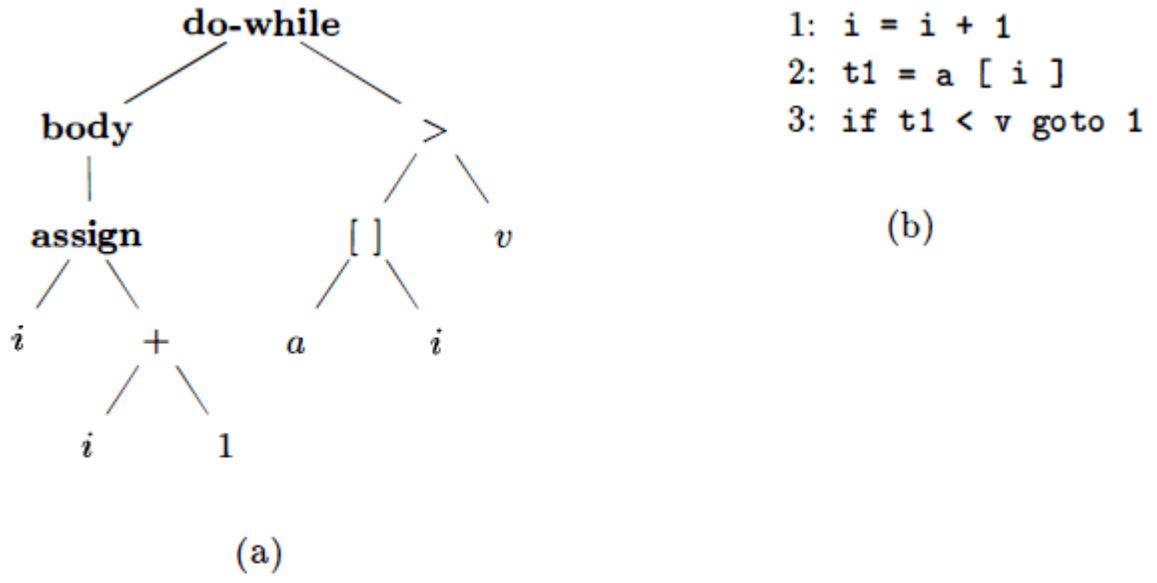


Figure 2.4: Intermediate code for "`do i = i + 1; while ( a[i] < v);`"

Sémantická analýza

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

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

Optimalizace

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

Generování kódu

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

Vícefázový a víceprůchodový překladač

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

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

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

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

Na strukturu překladače mají vliv:

- Vlastnosti zdrojového a cílového jazyka
- Vlastnosti hostitelského počítače

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

Testování a údržba překladače

Díky formální specifikaci jazyka je možné automatické provádění testů

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

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

Regulární gramatiky, regulární výrazy a konečné automaty

Thursday, May 30, 2013 8:36 AM

Regulární gramatiky

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 (jsou disjunktní).
- P je konečná množina odvozovacích pravidel. Každé pravidlo je tvaru

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

"cokoli poskládaný ze všech možnejch symbolů na cokoli"; S je prvek z N nazývaný počáteční symbol.

- **VĚTNÁ FORMA**
  Def.: Řetězec α se nazývá větnou formou v gramatice G, s počátečním symbolem S, platí-li:

  S =>* α, kde α ∈ (N ∪ T)*

- **VĚTA**
  Def.: Řetězec α se nazývá větou v gramatice G, s počátečním symbolem S, platí-li:

  S =>* α, kde α ∈ T*

- **FRÁZE**
  Def.: Nechť λ = α β γ je větná forma v gramatice G. Podřetězec β se nazývá frází větné formy λ vzhledem k neterminálnímu symbolu A, platí-li

  S =>* α A γ a A =>* β

  Tzn.frázi tvoří listy podstromu derivačního stromu.

- **JEDNODUCHÁ FRÁZE** větné formy α A γ vzhledem k neterm. A je podřetězec β, platí-li

  S =>* α A γ a A => β

- **L-FRÁZE**

  je nejlevější jednoduchou frází

Lineární gramatika = bezkontextová gramatika, která má nanejvýš jeden neterminál na pravé straně. Regulární gramatika je speciálním případem lineární gramatiky, kdy všechny neterminály jsou na levém konci/straně (levá lineární = levá regulární) nebo ekvivalentně pro pravou stranu.

Regulární gramatika – je to gramatika typu 3 = lineární, navíc převedená do regulárního tvaru (podle Chomského hierarchie). Pravidla těchto lineárních gramatik jsou omezena na jeden neterminál na levé straně. Pravá strana se u pravé regulární gramatiky skládá z jednoho terminálu (u lineární i z více), 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 regulární gramatiky**, které obsahují pravidla typu:

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

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

Lineární gramatika = má na pravé straně právě jeden neterminál

Regulární gramatika = gramatika, která popisuje regulární jazyk, přesně definovaný tvar pravidel (B -> a, B -> aC, B -> e pro pravou regulární gramatiku)

Regulární gramatika je tedy buď jen levá lineární gramatika nebo jen pravá lineární gramatika. Čistě lineární (levo-pravá) gramatika je pak taková gramatika, která sestává z pravých i levých pravidel současně.

Regular languages are also characterized by special grammars called regular grammars whose productions take the following form, where w is a string of terminals.
A → wB or A → w.
*Example. A regular grammar for the language of a*b* is*
S → Λ | aS | T
T → b | bT.
http://www.postech.ac.kr/~seungjin/courses/automata/handouts/handout04_4pp.pdf

http://web.cecs.pdx.edu/~jhein/lectures/Section.11.4.pdf

Grammar	Languages	Automaton	Production rules (constraints)
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$ (no restrictions)
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \gamma$
Type-3	Regular	Finite state automaton	$A \rightarrow a$ and $A \rightarrow aB$

From <http://en.wikipedia.org/wiki/Chomsky_hierarchy>

Regulární výrazy

Regulární výrazy umožňují algebraické manipulace s regulárními množinami - umožňují **vyjádření regulárních množin**. Třída regulárních výrazů nad abecedou Σ je definována takto:

- *e* a $\varnothing$ jsou regulární výrazy
- každé písmeno (symbol - znak) $\sigma \in \Sigma$ je regulární výraz nad Σ
- jsou-li R_1 a R_2 regulární výrazy nad Σ, pak i $(R_1 + R_2)$, $(R_1 . R_2)$ a R_1^* jsou regulární výrazy nad Σ

Daná množina je regulární množina nad Σ, právě když může být popsána vhodným regulárním výrazem nad Σ. Každý regulární výraz **U** popisuje jistou množinu **Ũ** slov nad **Σ**: **Ũ** $\subseteq$ **Σ***

Regulární množiny se vhodně charakterizují přechodovými grafy. Přechodový graf T nad abecedou Σ je konečný orientovaný graf, jehož každá hrana je pojmenována jistým slovem $w \in \Sigma^*$;alespoň jeden uzel je počáteční.

Množinu všech slov akceptovaných konečným automatem **A** označíme **Ã**. Množina je regulární nad **Σ** právě když je akceptována vhodným automatem nad **Σ**

Regulární výraz je řetězec popisující celou množinu řetězců (slov), konkrétně regulární jazyk. Používají se nejčastěji v počítačových programech a skriptovacích jazycích pro vyhledávání a úpravu textu. V případě, že uživatel chce v textu vyhledat nějaký řetězec, který nezná přesně nebo který může mít více variant, může zadat regulární výraz, který postihne všechny chtěné varianty. Program tak nalezne všechny části textu, které danému výrazu odpovídají.

Každý z regulárních výrazů označuje jistý regulární jazyk.

Kleeneho teorém: Každý regulární výraz je převoditelný na konečný automat.

Konečné automaty

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

S je konečná množina stavů.

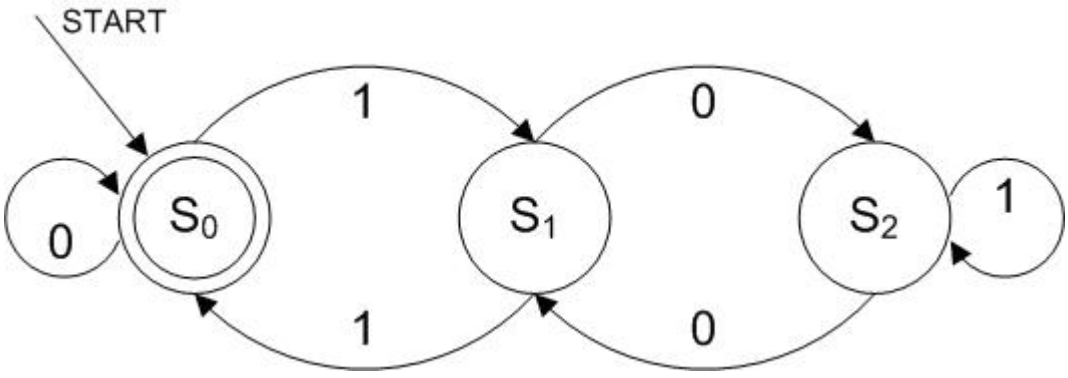
Σ (*velké sigma*) je konečná množina vstupních symbolů nazývaná abeceda.

P je tzv. přechodová funkce (též přechodová tabulka), formálně zobrazení $\delta: S \times \Sigma \rightarrow S$, popisující pravidla přechodů mezi stavy. Přechod je určen stavem ve kterém se automat nachází a symbolem, který přichází na vstup (nebo který je čten na vstupu)

s je počáteční stav (s naleží S)

F je množina koncových (přijímacích) stavů (F je podmnožinou 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.



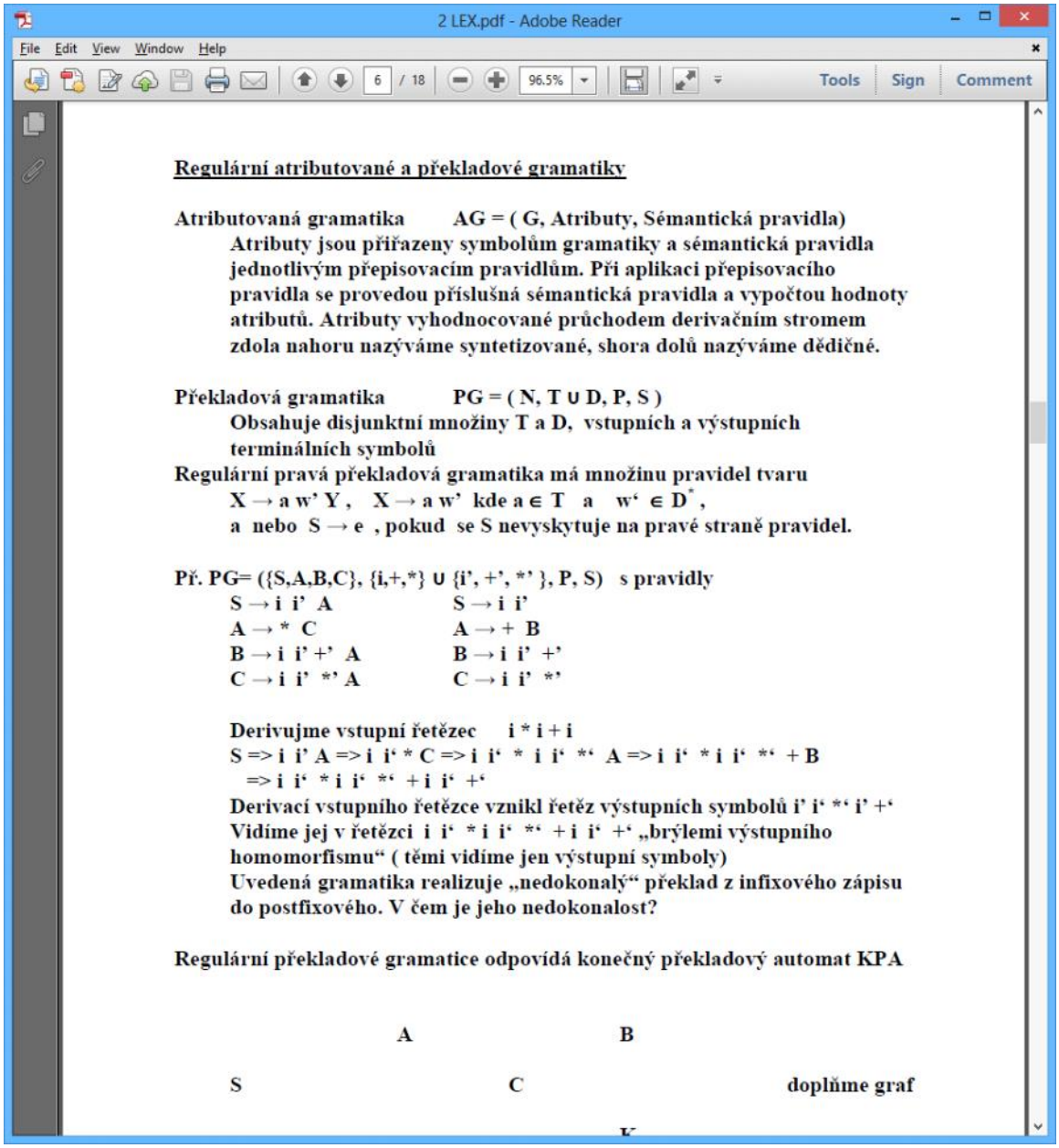
Determinismus

Konečné automaty se dělí dále na deterministické (DKA, DFA) a nedeterministické (NKA, NFA). U deterministických automatů, každý stav má právě jeden možný přechod pro každý možný vstup. U nedeterministického automatu jsou navíc povoleny e-hrany a více hran z jednoho stavu pro stejný vstup => v jeden okamžik se NKA může nacházet ve více stavech současně. Existuje však algoritmus, který umožňuje libovolný NKA převést na o něco složitější DKA (nejhůře v exponenciálním čase, prakticky však řádově rychleji).

Meze regulárních gramatik

Jak určit zdali je nějaký jazyk možné rozpoznat regulárním výrazem (konečným automatem, regulární gramatikou)? K tomuto lze použít tzv. Pumping teorém [wiki] nebo česky [abclinuxu].

Teorém vlastně říká, že v dostatečně dlouhém slově *w* daného regulárního jazyka můžeme nalézt tři části — *x*, *y* a *z*, přičemž nejdůležitější část *y* může zahrnovat i celé slovo. Aby byl tento jazyk regulární, musí platit, že část *y* můžeme ze slova vyjmout, nebo jí libovolně zopakovat, a přitom stále zůstáváme v rámci stejného jazyka.



Regulární atributované a překladové gramatiky

Atributovaná gramatika AG = (G, Atributy, Sémantická pravidla)

Atributy jsou přiřazeny symbolům gramatiky a sémantická pravidla jednotlivým přepisovacím pravidlům. Při aplikaci přepisovacího pravidla se provedou příslušná sémantická pravidla a vypočtou hodnoty atributů. Atributy vyhodnocované průchodem derivačním stromem zdola nahoru nazýváme syntetizované, shora dolů nazýváme dědičné.

Překladová gramatika PG = (N, T ∪ D, P, S)

Obsahuje disjunktní množiny T a D, vstupních a výstupních terminálních symbolů

Regulární pravá překladová gramatika má množinu pravidel tvaru

$X \rightarrow a\ w'\ Y$, $X \rightarrow a\ w'$ kde $a \in T$ a $w' \in D^*$,

a nebo $S \rightarrow e$, pokud se S nevyskytuje na pravé straně pravidel.

Př. PG= ({S,A,B,C}, {i,+,*} ∪ {i', +', *' }, P, S) s pravidly

$S \rightarrow i\ i'\ A$	$S \rightarrow i\ i'$
$A \rightarrow *\ C$	$A \rightarrow +\ B$
$B \rightarrow i\ i'\ +'\ A$	$B \rightarrow i\ i'\ +'$
$C \rightarrow i\ i'\ *'\ A$	$C \rightarrow i\ i'\ *'$

Derivujme vstupní řetězec i * i + i

S => i i' A => i i' * C => i i' * i i' *' A => i i' * i i' *' + B

=> i i' * i i' *' + i i' +'

Derivací vstupního řetězce vznikl řetěz výstupních symbolů i' i' *' i' +'

Vidíme jej v řetězci i i' * i i' *' + i i' +' „brýlemi výstupního homomorfismu" (těmi vidíme jen výstupní symboly)

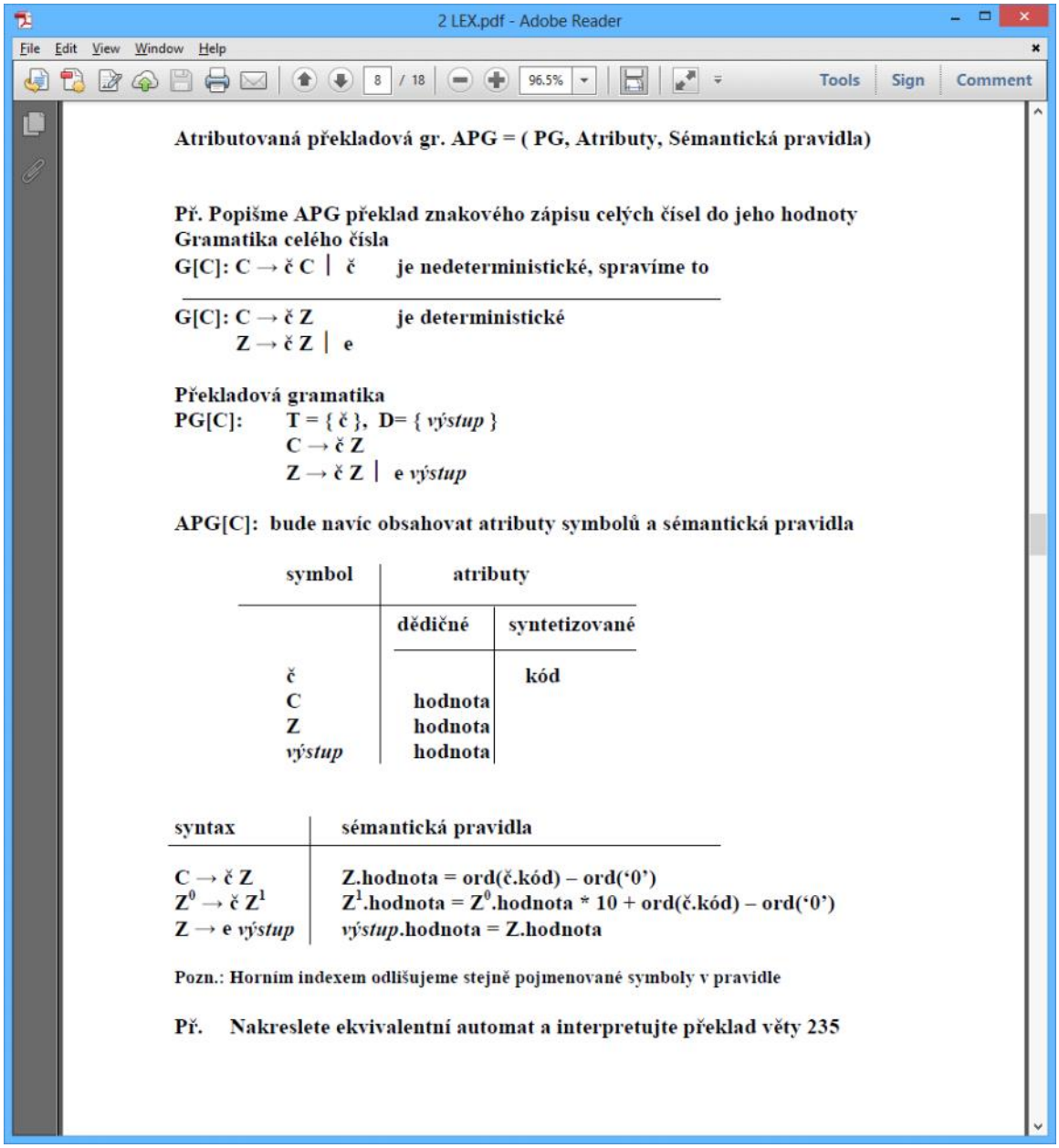
Uvedená gramatika realizuje „nedokonalý" překlad z infixového zápisu do postfixového. V čem je jeho nedokonalost?

Regulární překladové gramatice odpovídá konečný překladový automat KPA

A B

S C doplňme graf

K



Atributovaná překladová gr. APG = (PG, Atributy, Sémantická pravidla)

Př. Popišme APG překlad znakového zápisu celých čísel do jeho hodnoty
Gramatika celého čísla
G[C]: C → č C | č **je nedeterministické, spravíme to**

G[C]: C → č Z **je deterministické**
Z → č Z | e

Překladová gramatika
PG[C]: **T = { č }, D= { *výstup* }**
C → č Z
Z → č Z | e *výstup*

APG[C]: bude navíc obsahovat atributy symbolů a sémantická pravidla

symbol	atributy	
	dědičné	syntetizované
č		kód
C	hodnota	
Z	hodnota	
výstup	hodnota	

syntax	sémantická pravidla
$C \rightarrow č\ Z$	Z.hodnota = ord(č.kód) – ord('0')
$Z^0 \rightarrow č\ Z^1$	Z^1.hodnota = Z^0.hodnota * 10 + ord(č.kód) – ord('0')
Z → e *výstup*	*výstup*.hodnota = Z.hodnota

Pozn.: Horním indexem odlišujeme stejně pojmenované symboly v pravidle

Př. Nakreslete ekvivalentní automat a interpretujte překlad věty 235

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

Ekvivalence konečných automatů a regulárních gramatik

Thursday, May 30, 2013 8:38 AM

Regulární gramatiky popisují všechny *regulární jazyky* a v tomto smyslu (ve schopnosti popisu jazyka) jsou ekvivalentní s konečnými automaty a regulárními výrazy. Regulární gramatika ja buď pravá regulární (neterminály jsou vpravo) nebo levá regulární (neterminály jsou vlevo).

Regulární jazyk je formální jazyk (množina (i nekonečná) slov složených z omezené abecedy), který:

- může být akceptován deterministickým/nedeterministickým konečným stavovým automatem
- lze popsat regulárním výrazem
- lze ho generovat regulární gramatikou

Příkladem **ne**regulárního jazyka je $a^n b^n$, kde n > 1 (alespoň jedno *a* následované stejným počtem *b*), gramatika pro palindromy apod. - *lze určit na základě Nerodovy věty, která se užívá v důkazech, že nějaký jazyk není rozpoznatelný konečným automatem*

Každý regulární jazyk je rozpoznatelný konečným automatem; každý jazyk rozpoznatelný konečným automatem je regulární.

Kleenova věta: Libovolný jazyk je regulární, právě když je rozpoznatelný konečným automatem. Přechodový graf je T nad **S** je konečný orientovaný graf, jehož každá hrana je pojmenována jistým slovem w∈**S***. Alespoň jeden z uzlů grafu je počáteční a některé uzly jsou koncové. Ke každému přechodovému grafu T nad abecedou S existuje regulární výraz R nad S takový, že

$$\ddot{R} = \ddot{T}$$

a ke každému regulárnímu výrazu R nad S existuje konečný automat A takový, že

$$\ddot{A} = \ddot{R}$$

.

Postup převodu gramatiky na konečný automat

Potřebujeme získat gramatiku typu 3 ve standardní formě.

Regulární gramatika je ve **standardní formě**, jestliže obsahuje pouze pravidla tvaru $X \rightarrow aY$ a $X \rightarrow a$, $X \rightarrow e$ kde X,Y jsou neterminály, a je právě jeden terminál, e je prázdný symbol. Toho dosáhneme takto:

- Původní gramatika typu 3 (lineární): 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'*:
  - do *P'* zařadíme všechna pravidla z *P* ve tvaru $X \rightarrow aY$ a $X \rightarrow e$
  - za každé pravidlo $X \rightarrow x1\ x2\ x3\ Y$ zařadíme do *P'* soustavu pravidel:
    - $X \rightarrow x1\ X1$
    - $X1 \rightarrow x2\ X2$
    - $X2 \rightarrow x3\ Y$
      - za každé pravidlo $X \rightarrow z1\ z2$, zařadíme do *P'* soustavu:
    - $X \rightarrow z1\ Z1$
    - $Z1 \rightarrow z2\ Z2$
    - $Z2 \rightarrow e$

- Místo pravidel tvaru $X \rightarrow Y$ musíme zajistit to, aby z každého stavu X pro který máme $X \rightarrow Y$, bylo možné odvodit všechny řetězce, které lze odvodit z Y.
- N' vznikne obohacením N o všechny nově vytvořené neterminální symboly

Zkonstruujeme automat z nově vytvořené gramatiky

- stavy budou odpovídat neterminálním symbolům
- vstupy budou odpovídat terminálním symbolům
- přechodovou funkci zkonstruujeme na základě analogií
  - $X \rightarrow aY \leftrightarrow$ přechod ze stavu X do stavu Y při vstupu symbolu a
- počáteční stav bude odpovídat počátečnímu symbolu
- množinu koncových stavů určíme z pravidel $X \rightarrow e$

Tímto jsme získali **nedeterministický konečný automat**, který lze převést na **deterministický konečný automat.**

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

Nedeterministický a deterministický konečný automat

Thursday, May 30, 2013 8:38 AM

Deterministický konečný automat

Je 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 naleží S)
- F je množina koncových stavů (F je podmnožinou S)

Nedeterministický konečný automat

Nedeterministickým konečným automatem (NKA) *bez výstupu* nazýváme každou pětici

A = (Q, Σ, δ, S, F), kde:

- Q je konečná, neprázdná, množina stavů
- Σ je konečná neprázdná množina vstupních symbolů (vstupní abeceda)
- δ (přechodová funkce) je zobrazení δ: QxΣ → P(Q). Kde P(Q) je potenční množina (množina všech podmnožin množiny Q včetně prázné množiny *e*)
- S je množina počátečních stavů (S náleží Q) - není jednoznačně určen počáteční stav
- F je množina koncových stavů (F náleží Q)

Oborem hodnot přechodové funkce jsou všechny podmnožiny množiny stavů

Formálně je definován podobně jako DKA, ale obsahuje prvky nedeterminismu:

1. nejednoznačně určený počáteční stav (může jich být více)
2. nejednoznačné přechody (při přijetí stejného vstupu lze přejít do více stavů)
3. e - přechody (přechod do stavu bez přijetí vstupního symbolu)

- chování NKA lze popsat sekvencí pozic (množina stavů, ve kterých se automat může nacházet), z nichž každá jednoznačně definuje, zda je zpracovaný řetězec akceptován či zamítnut
- pozic je konečný počet
- přechody mezi pozicemi jsou jednoznačné
- Nejdůležitější rozdíl mezi DKA a NKA je v tom, že výsledkem přechodové funkce není pouze jeden stav, ale množina stavů, která může být i prázdná
- to vše jsou vlastnosti DKA a proto **ke každému NKA existuje ekvivalentní DKA**

  V případě nedeterministického konečného automatu (NKA) je vstupní slovo akceptováno (rozpoznáno,) pokud toto slovo může automat převést do některého z koncových stavů (množina F) z některého z počátečních stavů (množina S).

Převod NKA na DKA

1. Lineární gramatiku nejprve převedeme na regulární tvar (postup viz otázka [*Ekvivalence konečných automatů a regulárních gramatik*].

   2.Pak zkonstruujeme nedeterministický konečný automat a z něho nakonec deterministický (jak viz dále).

   A) Hlavní myšlenka je taková, že **každý stav vytvořeného DFA odpovídá množině stavů NFA.**

   B) Nebo: Z nedeterministického automatu se vytváří **strom**, který již popisuje deterministický automat, popisující tentýž problém.

Postup převodu

Samotný převod stojí na myšlence, že pokud lze ze vstupního uzlu

S
přejít jdo uzlu

A
a do uzlu

B
, tak vytvoříme nový uzel, říkejme mu

$[A,B]$
. Tento uzel bude mít stejné vstupy a výstupy jako sjednocení uzlů

A
a

B
. Nyní tabulka převedeného automatu obsahuje dva uzly

$\{S,[A,B]\}$
. Postup opakujeme pro uzel

$[A,B]$
. Takto postupně projdeme všechny stavy nově vytvářeného deterministického automatu.

Koncovými uzly převedeného deterministického automatu budou takové uzly, které jsou nadmnožinou koncových uzlů původního automatu (měl-li původně automat výstupní uzel

A
, tak uzel

$[A,X]$
, který vznikl jako sjednocení uzlu

A
a uzlu

X
, bude také výstupní).

Tento postup zároveň eliminuje všechny stavy, do kterých se deterministická verze automatu nemůže vůbec dostat. Zároveň ale mohou vzniknout uzly, které mají totožné vlastnosti (vstupní a výstupní uzly, konečnost, vlastnost *být počátečním uzlem*). Tyto uzly můžeme po doběhnutí algoritmu ztotožnit.

Příklad

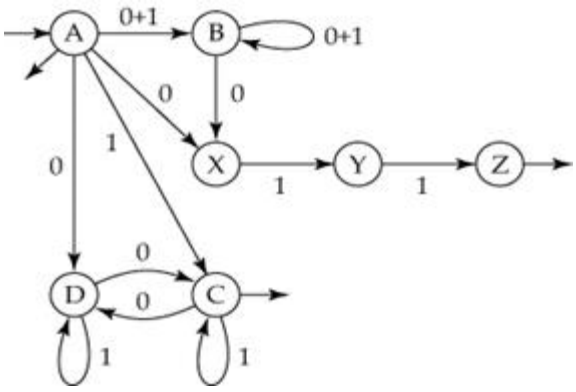
Zadaná pravá lineární gramatika:

```
A --> B | C
B --> 0B | 1B | 011
C --> 0D | 1C | e
D --> 0C | 1D
```

Pravá regulární gramatika:

```
A --> 0B | 1B | 0X | 0D | 1C | e
B --> 0B | 1B | 0X
X --> 1Y
Y --> 1Z
Z --> e
C --> 0D | 1C | e
D --> 0C | 1D
```

Nedeterministický konečný automat:



Deterministický konečný automat:

Přechodovou tabulku deterministického konečného automatu vytvoříme z přechodového diagramu nedeterministického kon. automatu takto:

- Do prvního řádku tabulky zapíšeme počáteční stav automatu a postupně zjistíme, do jakých množin stavů se nedeterministický automat může dostat z tohoto stavu přijmutím jednotlivých symbolů jeho vstupní abecedy.
- Z nalezených množin s více než jedním stavem vytvoříme tzv. *kompozitní* stavy det. automatu. Ty pak použijeme do přechodové tabulky det. automatu jako výstupy přechodové funkce pro počáteční stav a odpovídající vstupní symboly.
- Vzniklé kompozitní stavy (a případně i normální stavy) také využijeme v dalších řádcích přechodové tabulky a případně doplňujeme nové kompozitní stavy, do kterých se můžeme dostat z množin původních stavů každého kompozitního stavu přes vstupní symboly.
- Takto postupně vytvoříme celou přechodovou tabulku ekvivalentního deterministického automatu.
- Kompozitní stavy, zahrnující původní koncové stavy, můžeme označit také jako koncové.

	stav	0	1
<-->	A	BXD	BC
	BXD	BXC	BYD
<--	BC	BXD	BC
<--	BXC	BXD	BYC
	BYD	BXC	BZD
<--	BYC	BXD	BZC
<--	BZD	BXC	BD
<--	BZC	BXD	BC
	BD	BXC	BD

Nové stavy jsou A,BXD, BC, BXC, atd. Přechody 0,1.

Lexikální analýza, princip činnosti

Thursday, May 30, 2013 8:38 AM

Úkoly lexikálního analyzátoru

- Čtení zdrojového textu,
- Nalezení a rozpoznání lexikálních symbolů ve volném formátu textu, včetně případného rozlišení klíčových slov a identifikátorů. Vyžaduje spolupráci se syntaktickým analyzátorem.
- Vynechání mezer a komentářů,
- Interpretace direktiv překladače,
- Uchování informace pro hlášení chyb,
- Zobrazení protokolu o překladu.

Je prováděna **lexikálním analyzátorem**, který je vstupní a nejjednodušší částí překladače. Čte znaky zdrojového programu, a jeho výstupem jsou **tokeny**. Vstupní posloupnost znaků - program - je slučována do lexikologicky smysluplných mnohoznakových jednotek, tzv. ***lexémů*** *(*např. if, foo123bar). Tokeny pak symbolicky reprezentují lexémy (např. *if* pro lexém klíčové slovo *if*, *id* pro identifikátor *foo123bar*) a lexémy jsou tak vlastně jejich instance. Podoba lexémů reprezentujících jednotlivé tokeny je vymezena **vzorem** (**pattern**), typicky regulárním výrazem.
Kromě toho je jeho úkolem odstranění komentářů a eliminace přebytečných bílých znaků.

Token Name	popis (v podstatě pattern)	příklad lexémů	hodnota atributu
if	znaky i, f	if	-
else	zn. e, l, s, e	else	-
id	písmeno násl. písm./číslicí	foo, score, myId	pointer do tab. ~~Záznamů~~ symbolů
number	jakákoliv číselná konstanta	3.14, 10	pointer do tab. záznamů
literal	vše v uvozovkách	“hello world”	pointer do tab. záznamů
relop	<, <=, >, >=, =, <>	<, <=, >, >=, =, <>	LT, LE, GT, GE, EQ, NE

Token je tvořen dvěma částmi – názvem tokenu (token name) a hodnotou atributu (attribute value). Názvy tokenu jsou často abstraktní symboly, které jsou pak použity parserem pro syntaktickou analýzu. Jde např. o nějaké klíčové slovo nebo o soubor znaků představujících identifikátor. Operátory, klíčová slova a další ve skutečnosti atributové hodnoty nepotřebují. Pokud má token hodnotu atributu, jde o pointer do tabulky symbolů, která obsahuje dodatečné informace o tokenu, které nejsou součástí gramatiky.

Proud tokenů je předán parseru pro syntaktickou analýzu. Lexikální analyzátor také obvykle používá tabulku symbolů, do které ukládá objevené lexémy a ze kterých bere informace, aby mohl parseru podstrčit správný token. Jak název tokenu (typ - id, číslo,…), tak jeho atribut (číslo 0/1,…) ovlivňují rozhodování ve fázi parsování a pozdějších fázích. Parser proto potřebuje od analyzátoru dostat další token ke zpracování včetně informací z tabulky symbolů (viz obrázek níže).

V lexikální analýze mohou nastat nejednoznačnosti, pokud je jeden symbol prefixem jiného symbolu (== apod.). Pak se hledá nejdelší symbol a je vyžadována nápověda od syntaktického analyzátoru.

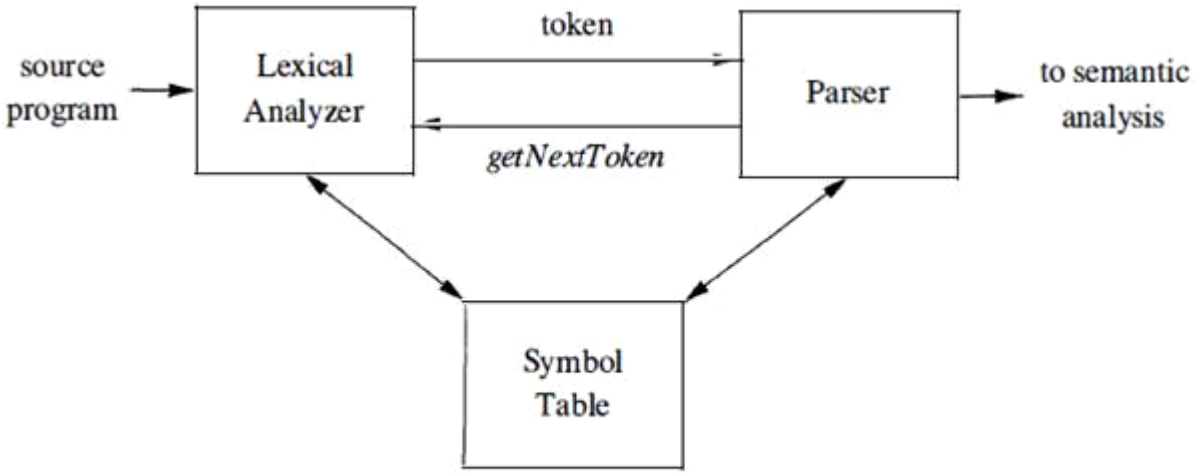


Figure 3.1: Interactions between the lexical analyzer and the parser

Často je potřeba dopředu skenovat vstup, aby se zjistilo, kde následující lexém končí. Proto lex. analyzátory typicky bufferují vstup.

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

Konstruktory lexikálních analyzátorů

Thursday, May 30, 2013 8:39 AM

Lex

Program LEX (Lexical Analyzer Generator) 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.

Lex umožňuje vytvořit lexikální analyzátor uvedením regulárních výrazů, které popisují vzory (patterns) pro tokeny. Vstupní notace pro Lex se nazývá *Lex language* a samotný nástroj je *Lex compiler*. Kompilátor Lexu transformuje vstupní vzory do přechodového diagramu (Jádrem toho všeho je *konečný automat*.) a generuje kód do souboru lex.yy.c, ve kterém je simulován přechodový diagram.

Vstupem Lexu je soubor, obvykle s koncovkou .l, např. *lex.l*, je napsaný v jazyce Lexu a popisuje lexikální analyzátor k vygenerování (rozpoznávaná slova a akce, které se mají po jejich rozpoznání provést). Slova (tokeny) 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.

Lex kompilátor překlopí lex.l do programu v C, který je vždy uložen v souboru *lex.yy.c*. Pak je tento soubor zkompilován vždy do *a.out*. Výstupem je fungující lexikální analyzátor, který bere proud vstupních znaků a vytváří z nich proud tokenů.

Hodnoty atributů (= numerický kód/pointer do tabulky symbolů/nic) jsou umístěny v globální proměnné `yylval`, kterou sdílí lexikální analyzátor a parser.

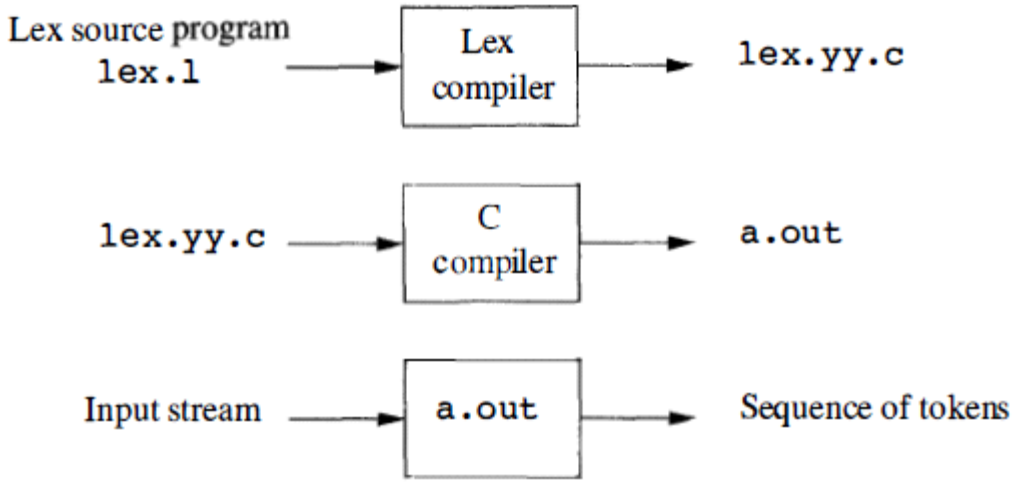


Figure 3.22: Creating a lexical analyzer with **Lex**

Struktura programu v Lexu

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

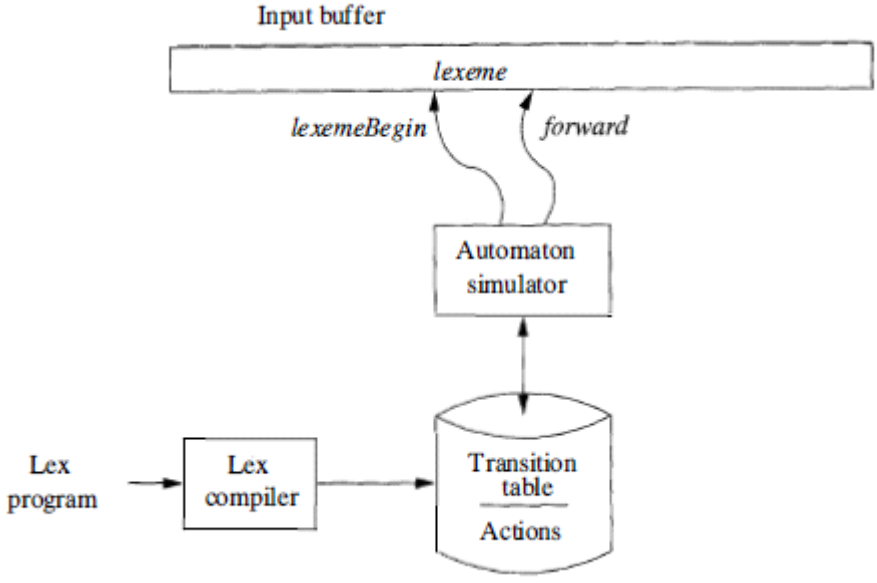
Příklad: program, který ve vstupním souboru nahradí všechny identifikátory slovem "IDENTIFIKATOR"

```
%%
[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

Architektura lexikálního analyzátoru generovaného Lexem



Lex z definovaných regulárních výrazů ze vstupního souboru → NKA → DKA; pravidlo: v případě konfliktů přiřazuje lexém vzoru dle nejdelšího prefixu.

Další varianty

- Flex – volně dostupná implementace Lexu, pro C.
- JLex – volně dostupná implementace Lexu, pro Javu.
- C# LEX - varianta JLex pro C#.

- PLY - implementace Lexu v Pythonu

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

Bezkontextové gramatiky a zásobníkové automaty, formální popis, ekvivalence

Thursday, May 30, 2013 8:39 AM

https://class.coursera.org/compilers/lecture/38

Bezkontextové gramatiky

Bezkontextové gramatiky (BKG) jsou gramatiky typu 2 podle Chomského hierarchie.

Gramatika je bezkontextová, **tvoří-li levou stranu všech přepisovacích pravidel právě jeden neterminální symbol**

Skládají se tedy z pravidel

$$A \to \gamma$$

kde A je právě jeden neterminál a

$$\gamma$$

je řetězec terminálů a neterminálů. Pravidlo

$$S \to e$$

je povoleno, pokud se S nevyskytuje na pravé straně žádného pravidla. Jazyky generované touto gramatikou jsou **rozpoznatelné nedeterministickým zásobníkovým automatem**.

Skládají se z **terminalů, neterminálů, počátečního symbolu a přepisovacích (produkčních) pravidel; G = (N,T,P,S)**.

- T = Terminály – jde o názvy tokenů (klíčová slova *if*, *else*, symboly „(“, „)“ atd.)
- N = Neterminály – syntaktické proměnné, pomáhají definovat jazyk generovaný gramatikou; zavádějí hierarchickou strukturu jazyka, která je klíčová pro syntaktickou analýzu
- S = Počáteční symbol – jeden z neterminálů
- P = přepisovací pravidla - ve tvaru

$$A \to \gamma, \text{kde } A \in N \text{ a } \gamma \in N \cup T$$

Příklad

Jazyk

$$L = \{0^n 1^n\}$$

pro

$$n \geq 0$$

, takovýto jazyk není rozpoznatelný konečným automatem, zásobníkovým ano („konečný automat neumí počítat“). U programovacích jazyků by to třeba znamenalo, že není možné závorkovat (vnořovat kód) do libovolné úrovně.

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,δ, 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

  $Qx(T \cup \{e\})xG^* \text{ do } QxG^*$
- 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.

  Zásobníkový automat se od konečného automatu liší ve dvou směrech:

1. Využívá vršek zásobníku při rozhodování jaký přechod provést.
2. Může manipulovat se zásobníkem jako součást provádění přechodu.

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 (q0, w, z0).

Příklad akceptace řetězce zásobníkovým automatem: *viz cvičení 10 (LL gramatiky) na courseware FJP*

Vztah bezkontextových gramatik a zásobníkových automatů

Zásobníkové automaty jsou ekvivalentní bezkontextovým gramatikám: pro každou bezkontextovou gramatiku existuje zásobníkový automat, který generuje (akceptuje) identický jazyk generovaný touto gramatikou a naopak.

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

1. 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,
   - q0 = q, počáteční stav automatu je q, neboť automat jiné stavy nemá,
   - z0 = S, tj. na počátku je v zásobníku startovací symbol gramatiky

- F = {}, což se interpretuje jako "automat akceptuje vyprázdněním zásobníku".
- *LL(k) gramatiky*

2. **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
   - δ je dáno rozkladovou tabulkou,
   - q0 = q,
   - z0 = #,
   - F = {r}.
   - *gramatiky: LR, SLR, LALR*

Nedeterministický syntaktický analyzátor.

Thursday, May 30, 2013 8:40 AM

Při syntaktické analýze konstruujeme derivační strom. Podle toho, jak je konstruován derivační strom věty, rozlišujeme dvě základní metody syntaktické analýzy:

1. metoda shora dolů

- derivační strom konstruujeme od kořene k listům a zleva doprava (provádíme levou derivaci)

2. metoda zdola nahoru

- postupujeme od listů směrem ke kořeni, ale také zleva doprava (provádíme pravou derivaci)

K syntaktické analýze se využívají zásobníkové automaty (ZA), které jsou obecně nedeterministické (nepoužitelné pro SA). Pro konstrukci SA lze použít buď:

- Deterministickou simulaci nedeterministického ZA = algoritmus syntaktické analýzy s návraty.
- Zdokonalit konstrukci ZA tak, aby byl pro určitou třídu BKG deterministický (pohled do zásobníku nebo dále do vstupního řetězce).

Obecný popis nedeterminismu a determinismu

Základem **nedeterminismu** je tedy vždy problém **výběr správného pravidla**, ať už analýzou shora dolů nebo zdola nahoru. Pokud se nemá analyzátor podle čeho rozhodnout, prostě **prochází prostor všech řešení** buď do šířky nebo do hloubky (backtracking) a hledá to správné řešení. Tato metoda je tedy značně **neefektivní**, protože v nejhorším případě může projít všechny možnosti a nenajít žádné správné řešení, tedy správnou množinu pravidel, jejichž expanzí/redukcí lze dosáhnout požadovaného výsledku.

V případě, že chceme analyzovat vstup **deterministicky**, musíme analyzátor **zdokonalit**. Ten musí mít přesnou informaci o tom, jaké pravidlo gramatiky v danou chvíli použít. Automat může využít informaci o **dosud provedené částečné derivaci** a také o **vstupu**, který ještě nebyl zpracován. Zásobníkovému automatu, který je abstraktním modelem syntaktického analyzátoru, tedy připravíme rozkladovou tabulku (lookup table), ve které bude určeno, jaké pravidlo má analyzátor použít podle vstupu a stavu zásobníku.

Metoda shora dolů

Postup z přednášek:

A.

Konstrukce ZA, který je modelem syntaktické analýzy metodou shora dolů.

$\mathcal{P} = (\{q\}, T, N \cup T, \delta, S, \emptyset)$, kde δ je definováno takto:

1. $\delta(q, e, A) = \{(q, \alpha) : A \to \alpha \in P\}$ pro $\forall A \in N$,
2. $\delta(q, a, a) = \{(q, e)\}$ pro $\forall a \in T$.

Operaci 1. nazýváme expanzí (nahradí na vrcholu zásobníku a tím i ve větné formě neterminální symbol některou jeho pravou stranou).

Operaci 2. nazýváme srovnáním (čteného vstupního symbolu a symbolu z vrcholu zásobníku).

Tento ZA má vrchol zásobníku vždy vlevo.

Př. Zapsat $\mathcal{P}$ pro G[E] (na tabuli)

$\mathcal{P} = (\{q\}, \{(,),+,*,a\}, \{E,T,F,(,),+,*,a\}, \delta, q, E, \emptyset)$
$\delta(q, e, E) = \{(q, E+T), (q, T)\}$
$\delta(q, e, T) = \{(q, T*F), (q, F)\}$
$\delta(q, e, F) = \{(q, (E)), (q, a)\}$
$\delta(q,a',a') = \{(q, e)\}$ pro $\forall a' \in \{(,),+,*,a\}$

Např. zpracování věty a+a
Tady je vrchol zásobníku
$(q, a+a, E) \vdash (q, a+a, E + T) \vdash (q, a+a, T + T)$ to jsou expanze
$\vdash (q, a+a, F + T) \vdash (q, a+a, a + T)$ teď provedeme 2krát srovnání
$\vdash (q, +a, + T) \vdash (q, a, T)$ a opět expandujeme
$\vdash (q, a, F) \vdash (q, a, a)$ a naposledy srovnáme $\vdash (q, e, e)$

Zásobník je po přečtení vstupního řetězce prázdný, takže řetězec byl akceptován

Derivační strom konstruujeme od kořene (ohodnoceného startovním symbolem) dolů k listům, zleva doprava podle levé derivace. Jedná se o zásobníkový automat LL. Počáteční konfigurace automatu se dá popsat uspořádanou trojicí **(q, w, alfa)**, kde:

q =vnitřní stav
w = dosud nezpracovaní část vstupu
alfa = obsah zásobníku
Na počátku práce je automat v konfiguraci **(q_0, w, z_0)**, napr. (q, abaaab, s).

Pokud jen generujeme větu v gramatice, můžeme v případě více pravidel se stejnou levou stranou náhodně vybírat. Naším úkolem však bývá spíše analýza již existující věty. Zde již náhoda nepřipadá v úvahu, protože posloupnost pravidel pro levou derivaci již nemusí být jednoznačná. Potřebujeme automat, který tuto analýzu provádí, a tento automat musí mít možnost jednoznačně vybírat mezi pravidly to správné.

Existují dva postupy analýzy (nedeterministická a deterministická):

- **analýza s návratem** (nedeterministická)
  - 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 atd. Tato metoda je sice účinná. ale zbytečně pomalá.
  - Rekurzivní sestup.

- **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 kontrolujeme obsah zásobníku (nestačí nám pouze vidět ten symbol, který ze zásobníku vyjímáme, ale i další, které jsou pod ním).
  - Prediktivní LL parser

Metoda zdola nahoru

Postup z přednášek:

Konstrukce ZA, který je modelem syntaktické analýzy metodou zdola nahoru.

$P = (\{q, r\}, T, N \cup T \cup \{\#\}, \delta, q, \#, \{r\})$, kde δ je definováno takto:

1. $\delta(q, a, e) = \{ (q, a) \}$ pro $\forall a \in T$,
2. $\delta(q, e, \alpha) = \{ (q, A) : A \to \alpha \in P \}$,
3. $\delta(q, e, \#S) = \{ (r, e) \}$.

Operaci 1. nazýváme přesun (přesun vstupního symbolu na vrchol zásobníku).

Operaci 2.nazýváme redukce (náhrada pravé strany pravidla na vrcholu zásobníku a tím i ve větné formě stranou levou).

Operace 3. je přijetí.

Tento ZA má vrchol zásobníku vpravo.
Konfiguraci budeme zapisovat ve tvaru: (stav, zásobník, vstup).
Zřetězením stavu zásobníku se zbytkem vstupu pak uvidíme jednotlivé větné formy

Př. Zapsat $\mathcal{P}$ pro G[E] (na tabuli)

- $\mathcal{P} = (\{q,r\}, \{(,),+,*,a\}, \{\#,E,T,F,(,),+,*,a\}, \delta, q, \#, r)$

  $\delta(q, a', e) = \{ (q, a') \}$ pro $\forall a \in T$,
  $\delta(q, e, E+T) = \{ (q, E) \}$
  $\delta(q, e, T) = \{ (q, E) \}$
  $\delta(q, e, T*F) = \{ (q, T) \}$
  …
  $\delta(q, e, \#E) = \{ (r, e) \}$

Např. zpracování věty a+a

Vrchol

$(q, \#, a+a) \vdash (q, \#a, +a) \vdash (q, \#F, +a) \vdash (q, \#T, +a)$
$\vdash (q, \#E, +a) \vdash (q, \#E+, a) \vdash (q, \#E+a, e) \vdash (q, \#E+F, e)$
$\vdash (q, \#E+T, e) \vdash (q, \#E, e) \vdash (r, e, e)$

ZA konstruované dle A. i B. jsou obecně nedeterministické (nepoužitelné pro SA). Pro konstrukci SA lze použít buď:

a) Deterministickou simulaci nedeterministického ZA = algoritmus syntaktické analýzy s návraty.
b) Zdokonalit konstrukci ZA tak, aby byl pro určitou třídu BKG deterministický.

Pozn.: Obsah zásobníku zřetězený se zbytkem vstupu je větnou formou.

- Konstruujeme derivační strom zdola od listů nahoru ke kořeni, přičemž postupujeme zleva doprava.

- Stejně jako u první metody i zde budeme používat lineární rozklad, tentokrát pro pravou derivaci - je to proces nalezení pravého rozkladu věty (LR gramatika).
- I zde musíme rozhodovat, která pravidla 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** (nedeterministicky)
  - Vybereme ve větné formě jeden podřetězec (jako první vybíráme ten, který začíná nejvíc nalevo, je co nejdelší a je shodný s pravou stranou některého pravidla), přepíšeme neterminálem na pravé straně pravidla a pokračujeme v konstrukci derivačního stromu.
  - Pokud zjistíme, že tento krok nevede k úspěchu, vyzkoušíme jiný podřetězec atd (backtracking). Tato metoda je příliš časově náročná.
- **deterministická analýza (LALR, SLR)**
  - Využíváme další informace získané při překladu, např. obsah nepřečtené části vstupního kódu nebo obsah zásobníku.

ZA konstruované výše uvedenými metodami jsou obecně nedeterministické (nepoužitelné pro SA). Pro konstrukci SA lze použít buď:

a) Neterministickou simulaci nedeterministického ZA = algoritmus syntaktické analýzy s návraty.
b) Zdokonalit konstrukci ZA tak, aby byl pro určitou třídu BKG deterministický.

Pozn.: Obsah zásobníku zřetězený se zbytkem vstupu je větnou formou.

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

Derivace a derivační strom, víceznačnost gramatiky

Thursday, May 30, 2013 8:40 AM

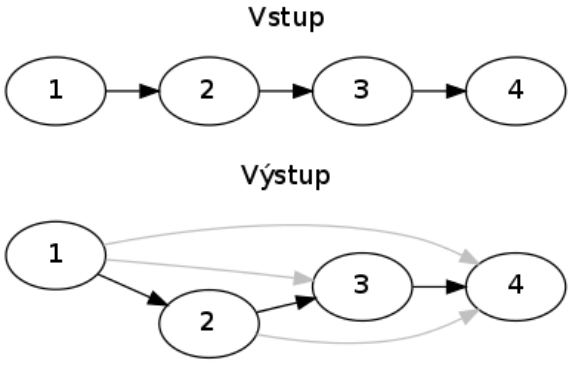
Derivace

- Posloupnost kroků odvození terminálu pomocí přepisovacích pravidel gramatiky
- Derivační pohled odpovídá konstrukci parsovacího (syntaktického) stromu shora dolů (top-down).
- Parsování zdola nahoru (bottom-up) je spjato s pravými derivacemi.
- Podle toho, který neterminál nahradit v každém kroku derivace, se rozlišují **levá a pravá derivace**.

DERIVACE řetězce α je posloupnost kroků odvození α pomocí přepisovacích pravidel gramatiky

$S = \alpha_1 => \alpha_2 => \ldots => \alpha_n = \alpha$

Dtto $S =>^* \alpha$ pozn.: $=>^*$ je uzávěr relace => (všechny přechody kam se dá transitivně dostat)



PŘÍMÁ DERIVACE: α A β => α γ β ,kde A -> γ ϵ P (pozn.: P je množina pravidel, pomocí kterých lze odvodit jazyk.)

Derivační strom

Derivační strom (parse tree) je orientovaný acyklický graf a je grafickou reprezentací, která říká, v jakém pořadí byla přepisovací pravidla uplatňována na neterminály, tedy jak vznikla věta jazyka.

- Kořen stromu je označen startovacím symbolem gramatiky
- Každý vnitřní uzel je ohodnocen neterminálními symboly.
- Listy jsou ohodnoceny terminálními symboly.
- Listy se čtou zleva doprava a dávají větu (generovanou gramatikou).
- Jestliže uzly $n_1, n_2, \ldots, n_k$ jsou bezprostřední následníci uzlu n, jsou ohodnoceny symboly $A_1, A_2, \ldots, A_k$ a uzel *n* je ohodnocen A, pak v množině pravidel gramatiky existuje pravidlo $A \rightarrow A_1A_2 \ldots A_k$.
- Není třeba značit orientaci hran.

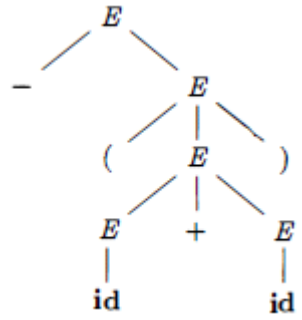
Jiná definice jazyka generovaného gramatikou je množina vět, které mohou být vytvořeny derivačním stromem. Proces hledání derivačního stromu pro danou větu (řetězec terminálů) se nazývá **parsování** tohoto řetězce

Derivační strom ignoruje variace v pořadí, v jakém jsou symboly přepisovány. Proto je mezi derivacemi a derivačními stromy vztah 1:N – např.:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

Jsou různé derivace, jejich derivační strom ale vypadá stejně:



Pro získání jednoznačného derivačního stromu pro derivaci se proto užívá buď pravá a nebo levá derivace.

Víceznačnost gramatik

Gramatika, která generuje větu, pro níž lze sestavit aspoň dva různé derivační stromy, je víceznačná. Jinak řečeno: je to taková gramatika, která produkuje více než jednu levou nebo víc než jednu pravou derivaci pro tutéž větu.

Příklad: Pro gramatiku

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

umožňuje vytvořit dvě levé derivace pro id + id * id (násobení není upřednostněno před sčítáním):

$$\begin{aligned} E &\Rightarrow E+E \\ &\Rightarrow \mathbf{id}+E \\ &\Rightarrow \mathbf{id}+E*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id} \end{aligned} \qquad \begin{aligned} E &\Rightarrow E*E \\ &\Rightarrow E+E*E \\ &\Rightarrow \mathbf{id}+E*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id} \end{aligned}$$

- **Nutnou podmínkou jednoznačnosti gramatiky je, aby pro žádný neterminální symbol neexistovalo jak pravidlo rekurzivní zprava, tak i pravidlo rekurzivní zleva**
- **Problém nejednoznačnosti bezkontextových jazyků je algoritmicky nerozhodnutelný.**

  Je potřeba buďto vytvořit jednoznačné gramatiky pro kompilaci aplikací, nebo u nejednoznačných gramatik zavést dodatečná pravidla, která řeší případné nejednoznačnosti.

Odstranění levé rekurze

- **Levorekurzivní gramatiku nelze použít k analýze shora dolů**

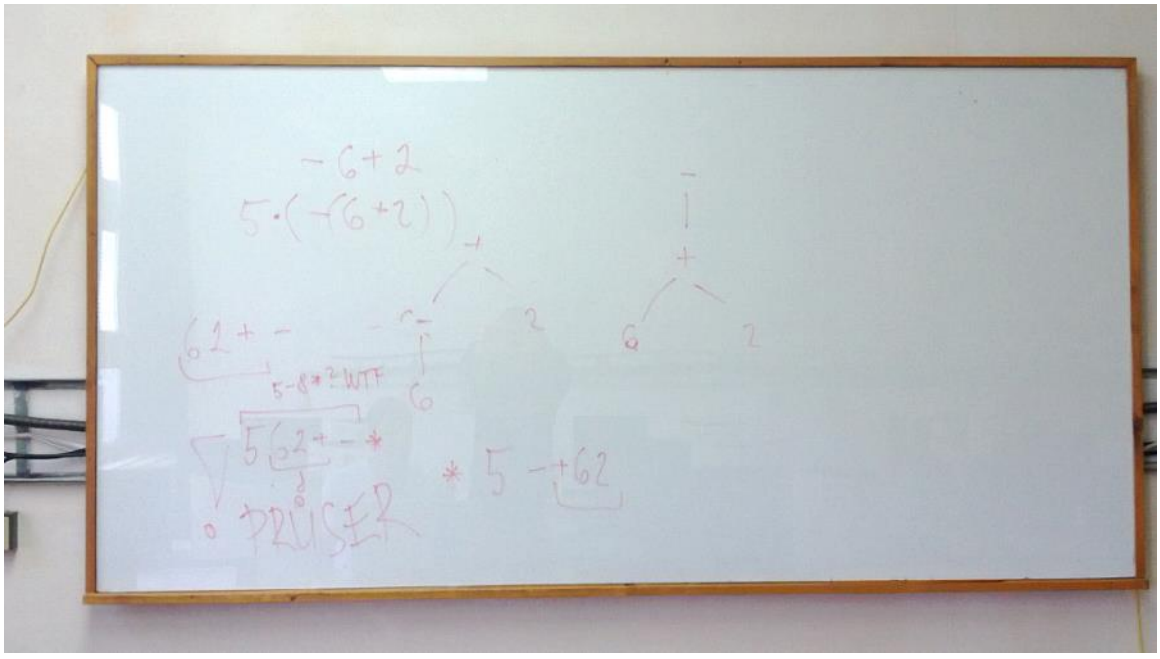
  Odstranění pravidla rekurzivního zleva:

- Nechť je dána BKG G = (N, T, P, S), ve které,
  - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
    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:
  - $A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \mid \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A'$

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


-6+2
5*(-(6+2))
61+-
5-8*2 WTF
5 6 2 + - *
* 5 - + 6 2
PRUSER

Deterministická syntaktická analýza

Thursday, May 30, 2013 8:41 AM

Např. s pomocí **prediktivního parseru** = rekurzivně sestupný parser, který nevyžaduje zpětné kroky. Je ho možné vytvořit jen pro LL(k) gramatiky, což jsou bezkontextové gramatiky, pro které existuje kladné *k*, které umožní rekurzivně sestupnému parseru rozhodnout se, které přepisovací pravidlo použít na základě *k* dalších načtených symbolů. LL(k) gramatiky vylučují mnohoznačnost a levou rekurzi. Jakákoliv bezkontextová gramatika může být transformována na ekvivalentní nelevorekurzivní gramatiku, ale odstranění levé rekurze ne vždy vede k LL(k) gramatice. Prediktivní parser běží v lineárním čase.

Deterministická syntaktická analýza využívá další informace získané při překladu – obsah zásobníku a obsah nepřečtených vstupních tokenů. Na základě těchto informací umožňují následující funkce vhodný výběr přepisovacích pravidel, a tím prediktivní parsování:

FIRST(A) = množina terminálů, kterými mohou začínat řetězce odvozené z A **(terminály na začátku A)**

Funkce first zjišťuje, co vznikne přepsáním jednotlivých neterminálů na levé straně všech pravidel.

- A => bxayB → b náleží FIRST(A)

Algoritmus výpočtu

- FIRST(A) pro A = terminál nebo *e:* je terminál/*e*
- Když je A neterminál:
  - Je to první terminál ve všech přepisovacích pravidlech s A na levé straně
  - Pokud jsou v přepisovacím pravidle na P straně jen neterminály, hledá se FIRST prvního neterminálu na pravé straně
  - Pokud lze nějaký z těchto neterminálů přepsat na prázdný řetězec *e*, je potřeba se podívat na *first* neterminálu následujícího po něm v některém z přepisovacích pravidel

FOLLOW(A) = množina terminálů, které mohou následovat za A v některé větné formě v derivacích **(terminály hned za A)**

- S => bxCAyZ → y náleží FOLLOW(A)

Algoritmus výpočtu

1. Polož FOLLOW (A) = θ

2. Je-li A počáteční symbol G, přidej e do FOLLOW(A)

3. Pro všechny pravé strany pravidel z G tvaru α A β přidej FIRST (β) do FOLLOW (A), nepřidávej ale e.

4. Je-li v G pravidlo L → α A nebo L → α A β , kde FIRST (β)

obsahuje e, pak přidej do FOLLOW (A) množinu FOLLOW (L)

Vytváříme vždy pro všechny neterminály zároveň!

$FIRST_k(A)$, $FOLLOW_k(A)$ = zobecnění na množiny terminálních řetězců o délce nejvýše *k*

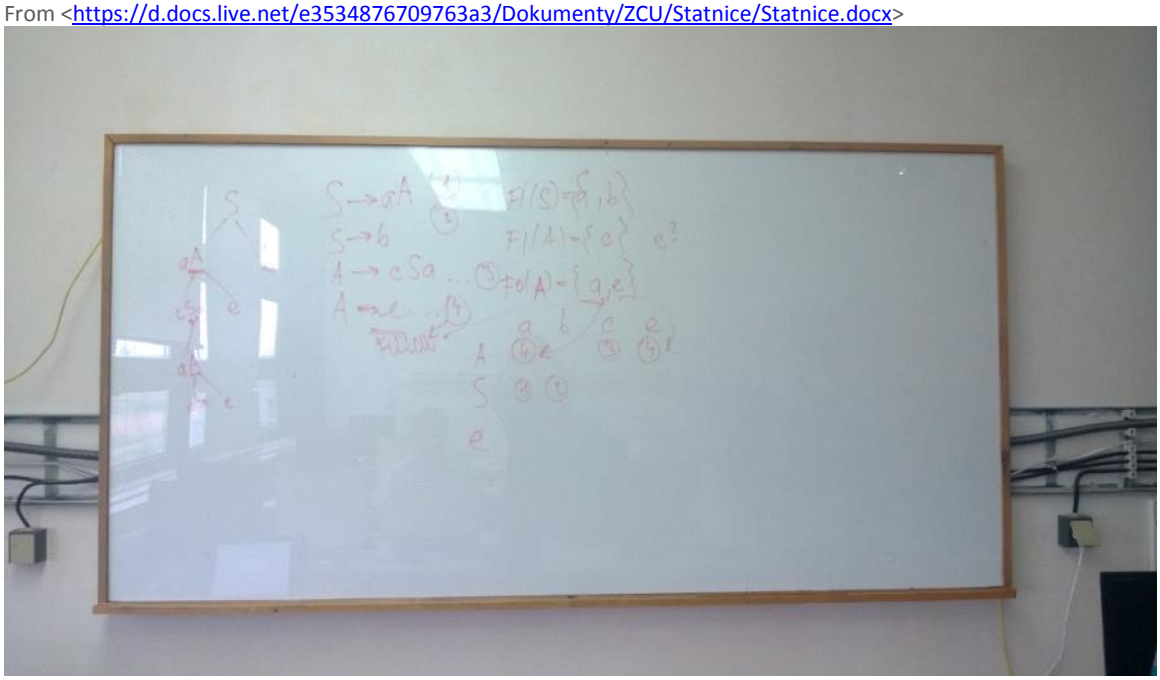
Tyto funkce slouží k vytvoření **rozkladové tabulky**, která nahrazuje přechodovou funkci. V první řádce jsou uvedeny všechny možné vstupy, v prvním sloupci všechny možné stavy vrcholu zásobníku

(vč. dna zásobníku #). Má stavy:

- **Srovnání (pop)** – na vstupu i na vrcholu zásobníku jsou stejné hodnoty
- **Přijetí (accept)** – bylo dosaženo dna zásobníku a přijímá se prázdný symbol *e*
- **expanze (expand)** – aplikace přepisovacího pravidla, které je uvedené v buňce tabulky určené vstupem (který sloupec) a vrcholem zásobníku (který řádek)
- **chyba (error)** – pokud pro vstup a hodnotu na vrcholu zásobníku je buňka v tabulce prázdná → vstupní řetězec není větou jazyka

Syntactic Analysis in terms of **bottom up algorithms: LR, SLR, LALR**

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



Rekurzivní sestup

Thursday, May 30, 2013 8:41 AM

https://class.coursera.org/compilers/lecture/25

Top Down

Shora

Zleva doprava

Rekurzivní sestup nebo také **rekurzivní sestupný parser** postupuje shora dolů a je sestaven ze vzájemně se volajících procedur. Každá taková procedura obvykle implementuje jedno přepisovací pravidlo gramatiky. (Kromě něj do top-down parserů patří prediktivní parsery založené na LL(k) gramatikách.)

Prediktivní parser je rekurzivně sestupný parser, který nevyžaduje zpětné kroky. Je ho možné vytvořit jen pro LL(k) gramatiky, což jsou bezkontextové gramatiky, pro které existuje kladné *k*, které umožní rekurzivně sestupnému parseru rozhodnout se, které přepisovací pravidlo použít na základě *k* dalších načtených symbolů. LL(k) gramatiky vylučují mnohoznačnost a levou rekurzi. Jakákoliv bezkontextová gramatika může být transformována na ekvivalentní nelevorekurzivní gramatiku, ale odstranění levé rekurze ne vždy vede k LL(k) gramatice. Prediktivní parser běží v lineárním čase.

Rekurzivní sestup s návratem je technika určování použitého produkčního pravidla zkoušením všech pravidel. Není limitován na LL(k) gramatiky, ale nemá zaručeno skončit, pokud gramatika není LL(k). Může vyžadovat exponenciálni čas pro svůj běh.

Princip

Hlavní myšlenka je taková, že pro každý neterminál gramatiky je implementována příslušná funkce v programu.

- každému neterminálnímu symbolu *A* odpovídá procedura *A*
- tělo procedur je dáno pravými stranami pravidel pro *A*
- pravé strany musí být rozlišitelné na základě symbolů vstupního řetězce
- je-li rozpoznána pravá strana, pak v případě neterminálního symbolu vyvolá *A* proceduru pro rozpoznaný neterminální symbol, v případě terminálního symbolu, ověří *A* jeho přítomnost ve vstupním řetězci a zajistí přečtení dalšího znaku ze vstupu
- rozpoznané pravidlo analyzátor oznámí (např. jeho číslo)
- chybnou strukturu vstupního řetězce oznámí chybovým hlášením

Terminál na pravé straně je porovnán s dalším vstupním symbolem. Pokud se shodují, přejde se na další vstupní symbol a na další symbol na pravé straně. V opačném případě je nahlášena chyba.

O neterminál na pravé straně je postaráno voláním příslušné funkce. Po jejím vykonání se pokračuje dalším symbolem na pravé straně.

Pokud na pravé straně už nejsou žádné symboly, funkce končí (function returns).

Takto se postupně volají všechny funkce, až se nakonec opět ocitneme ve funkci pro startovací symbol, která byla zavolána jako první. Ta také oznamuje úspěšný průběh, pokud se prošel celý vstupní řetězec.

```
    void A() {
1)      Choose an A-production, A → X1X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Pseudokód výše je nedeterministický, protože začíná volbou A-přepisovacího pravidla, které se používá blíže nepopsaným způsobem.

Příklad: http://info.lu2.name/soubory/prekl_06_604.pdf

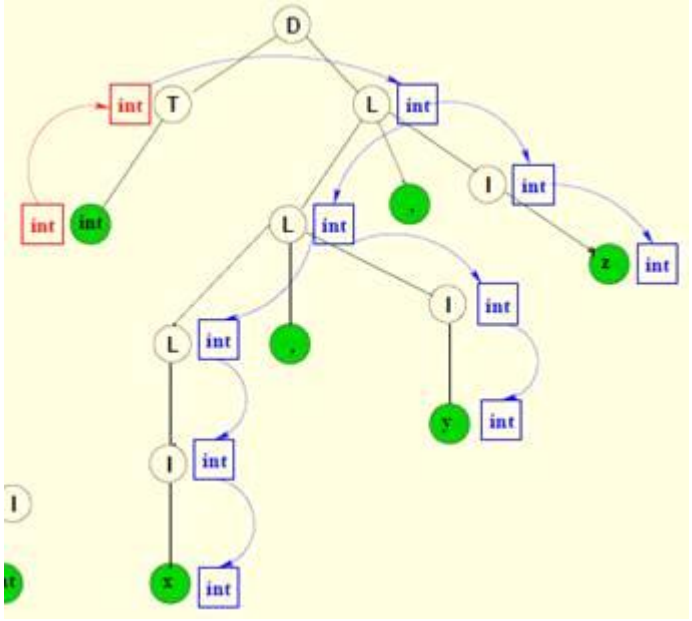
Obecně může rekurzivní sestup potřebovat backtracking, tzn. někdy je třeba se vrátit a opakovaně číst vstup. Backtracking je však potřeba zřídkakdy (dá se eliminovat volbou vhodné gramatiky - tedy zavedením jistých omezení). Kód výše neumožňuje backtracking, bylo by je nutno modifikovat – v řádce 7 se pak vrátit na řádku 1 a zvolit jiné pravidlo, popř. nahlásit chybu, když už žádné další nejde použít.

Sémantické zpracování

Při rekurzivním sestupu se může provádět také sémantické zpracování. Sémantické zpracování zahrnuje vyhodnocení atributů symbolů v derivačním stromu. Atributy = vlastnosti gramatických symbolů nesoucí sémantickou informaci (hodnota, adresa, typ, scope, spojitost mezi formálními a skutečnými parametry apod.).

Způsoby vyhodnocení:

1. procházením stromem od listů ke kořenu = **syntetizované atributy**
2. procházením stromem od rodiče k potomkovi, od staršího bratra k mladšímu = dědičné atributy (např vícenásobné deklarace v C – int x,y,z)



Je nutné doplnit procedury lex. analýzy (LA) i syntakt. analýzy (SA) takto:

- LA bude předávat s přečteným vstupním symbolem i jeho atributy.
- procedury SA pro neterminály doplnit o:
  - vstupní parametry odpovídající dědičným atributům
  - výstupní parametry odpovídající syntetizovaným atributům
  - zavést lokální proměnné pro uložení atributů pravostranných symbolů
  - před vyvoláním procedury korespondujícího neterminálu z pravé strany vypočítat hodnoty

jeho dědičných atributů
    - na konec procedury popisující pravou stranu pravidla zařadit příkazy vyhodnocující syntetizované atributy

Vlastnosti

- pro metodu rekurzivního sestupu, tj. analýza shora dolů, se používají *LL* gramatiky (levorekurzivní se nedají použít, protože by se program mohl zacyklit voláním pořád té samé procedury)
- jednoduchá *LL* gramatika je taková gramatika, kde levou stranu tvoří právě jeden neterminální symbol a kde každá pravá strana začíná terminálním symbolem
- navíc musí platit, že např. pro pravidla *A->...* jsou počáteční symboly různé
- obecná *LL* gramatika nemá omezení, ale musí pro ni existovat rozkladová tabulka

Principy a podmínky LL analýzy

Thursday, May 30, 2013 8:42 AM

LL-gramatika

LL gramatika je jakákoliv gramatika, z níž se dá udělat rozkladová tabulka pro LL parser. **LL(k) parser** se kouká při parsování věty na následujících *k* tokenů, aby věděl, co dál. Pokud takový parser může být použit pro nějakou gramatiku, aniž by se musel použít backtracking, jedná se o **LL(k) gramatiku**.

Aby se ze vstupní gramatiky dala udělat LL(1) gramatika – eliminace levé rekurze, levá faktorizace (eliminace překrývajících se množin FIRST, tj. "rozsekání" pravidel se stejným začátkem pravé strany na několik menších, už jednoznačných)

př: STAT => if EXP then STAT | if EXP then STAT else → STAT => if EXP then STAT ElsePart; ElsePart => else STAT | e)

Podmínky

- Nesmí být přítomna levá rekurze.
- Nesmí dojít k first-follow (u neterminálu, který se přepisuje na "e") kolizi, first-first kolizi

Třídy jazyků LL(k)

L = Left to right -> vstupní text (soubor) čteme zleva doprava

L = Left parse -> vytváříme levý rozklad

K = při rozhodování mezi pravidly potřebujeme vidět nejvýše *k* znaků z nepřečtené části vstupu

Tzn.: **LL(k)** gramatika provádí deterministický rozbor *čtením textu z **L**eva doprava*, s *použitím **L**evé derivace* a *prohlédnutí **k** dalších symbolů vstupního textu*.

- gramatika je typu **LL(k)**, jestliže ji lze použít pro deterministickou syntaktickou analýzu metodou shora dolů (tj. 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

LL(0) gramatika

- lze určit správné pravidlo aniž bychom předem potřebovali vidět nějaký znak na vstupu
- každý neterminál musí mít jen jednu jedinou pravou stranu (jen jedno přepisovací pravidlo)
- neumožňuje rekurzi
- prostě jen určují, jestli sekvence patří do jazyka nebo ne, žádné rozhodování není potřeba.
- LL(0) gramatiky jsou nevhodné pro popis programovacích jazyků, protože zde není možná rekurze a pro každý neterminál existuje právě jedno pravidlo (důsledek faktu, že gramatika se nemůže rozhodnout podle následujícího vstupního symbolu), tudíž mohou generovat jen jazyk s jediným slovem.
  From <http://cs.wiki pedia.org/wiki/LL_syntaktick%C3%BD_analyz%C3%A1tor>

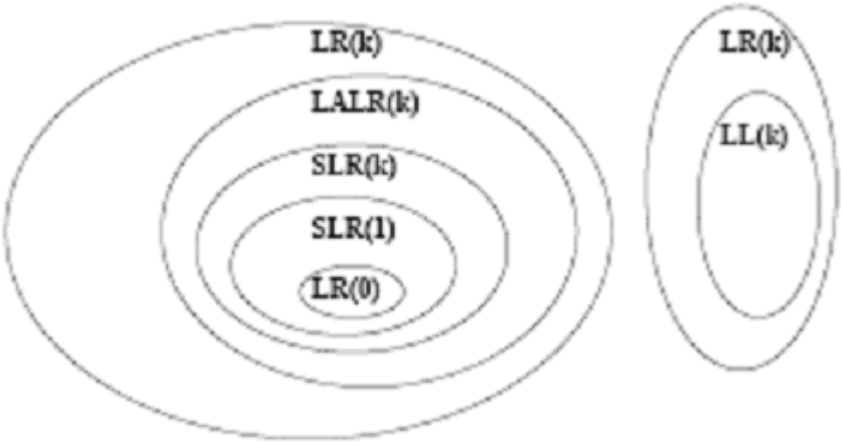
- Příklad:

```
G == id name lastname
id == [0-9]+
name == string
lastname == string
string == <unicode>+
```

LL(1) gramatika

- pro danou gramatiku *G* se vystačí při rozhodování o výběru pravidla pro expanzi s informací o dopředu prohlíženém řetězci délky 1 -> proto LL(1) je **gramatika silná**
- **jednoduchá LL (1) gramatika** je taková bezkontextová gramatika jestliže platí:
  - pravá strana každého pravidla začíná terminálním symbolem např. A -> aB
  - pokud mají 2 pravidla stejnou levou stranu, pak pravé strany začínají různými terminálními symboly např. A -> aB, A->bB
  - to znamená, že v každém políčku rozkladové tabulky bude právě jeden element
- **Obecná LL(1):**
  - gramatika nemá omezení, ale musí pro ni existovat rozkladová tabulka

Mohutnosti gramatik



Typy analýzy

- shora (top-down)
- sdola (bottom-up) (vyžadují LR gramatiku, takže se netýkají této otázky)

Analýza shora = analýza top-down

- Při hledání derivace začínáme počátečním symbolem a snažíme se dostat k hledanému slovu
- LL analýza: hledáme levou derivaci, vstupní slovo analyzujeme zleva
  - Přesně určuje volbu pravidel při analýze a umožňuje jednoznačný postup při odvození
  - Gramatika, která je jednoznačná a lze ji takto analyzovat: LL gramatika
  - Využívá se zásobníkový automat
- LL(k) označení gramatiky pro LL analýzu, číslo k určuje, kolik následujících symbolů na vstupu je nutné znát pro analýzu slova
- LL(1): nejpoužívanější gramatika, stačí znát jeden následující symbol
  - Jde vlastně o variantu rekurzivního sestupu bez backtrackingu
- LL(0): umožňuje jen jazyky s konečným počtem slov
- LL gramatiky s k>1 lze převést na LL gramatiky s k = 1
  - Existují přesné popisy, jak jednotlivá pravidla nahrazovat (přidávají se neterminály a pravidla se upravují, aby při analýze stačilo znát jeden další symbol)

LL parsery = parsery s analýzou top-down

LL parsery používají parsing **shora dolů**, zpracovávají vstup zleva doprava a konstruují nejlevější derivaci. Proto se také nazývá L (left-to-right) L(leftmost derivation). Občas se setkáváme s označením LL(k), kde k značí počet tokenů, které potřebujeme znát při rozhodování o průběhu další analýzy bez toho, aby bylo třeba používat backtracking (= prediktivní parser). Také se v této souvislosti používá pojem look-ahead. Prakticky do nedávné doby se tyto gramatiky příliš nepoužívaly, ovšem na počátku 90. let minulého století došlo ke změně přístupu.

Syntaktická analýza LL gramatik

Budeme se zabývat algoritmem syntaktické analýzy, který vytváří derivační strom analyzovaného řetězce směrem shora dolů. Základní princip syntaktické analýzy můžeme v tomto případě formulovat takto:

Je dána bezkontextová gramatika G= (N,T,P,S) a řetězec w = $a_1\ a_2 \dots a_n$, který je větou z L(G). Pak existuje levá derivace

$S = \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_n = w.$

Vzhledem k tomu, že derivace je levá, má každá větná forma γi tvar:

$\gamma i = a_1\ a_2 \dots a_j\ A_i\ \beta_i,$

kde $a_1, a_2 \dots, a_j$ jsou terminální symboly, A_i je neterminální symbol, β_i je řetězec terminálních a neterminálních symbolů. Přitom řetězec $a_1a_2 \dots a_j$ je předponou věty w, $j \geq 0$.

Podmínky LL analýzy

Předpokládejme, že $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ jsou všechna pravidla v P s neterminálním symbolem A na levé straně. Pak základní problém syntaktické analýzy metodou shora dolů spočívá v nalezení toho pravidla $A \rightarrow \alpha_k$, jehož aplikací dostaneme z větné formy γ_i větnou formu $\gamma_i + 1$.

Pro výběr pravidla $A \rightarrow \alpha_k$, je možno použít:

1. informaci o dosavadním průběhu (historii) analýzy,
2. informaci o dosud nepřečtené části vstupního řetězce (dopředu prohlíženém řetězci omezené délky).

Pokud tyto informace vždy stačí k jednoznačnému výběru pravidla $A \rightarrow \alpha_k$, pak se gramatika G nazývá *LL gramatika*. Název je odvozen od toho, že při čtení vstupního řetězce zleva je vytvářen levý rozklad. Při syntaktické analýze LL gramatik jsou do zásobníku ukládány řetězce, které odpovídají levým větným formám nebo takovým jejich příponám, které vzniknou odejmutím předpony tvořené řetězcem terminálních symbolů.

Základními operacemi syntaktického analyzátoru pro LL gramatiky (LL analyzátoru) jsou:

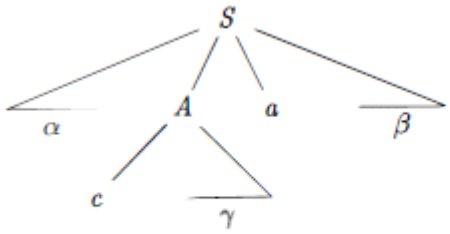
- **Expanze** – neterminální symbol na vrcholu zásobníku je nahrazen pravou stranou vybraného pravidla
- **Srovnání** – terminální symbol na vrcholu zásobníku se ze zásobníku vyloučí, jestliže je shodný se symbolem, který byl ze vstupního řetězce přečten.
- **Přijetí** – vstupní řetězec je přečten a zásobník je prázdný.
- **Chyba** – ve všech ostatních případech.

Pokud pro danou gramatiku G vystačíme při rozhodování o výběru pravidla pro expanzi s informací o dopředu prohlíženém řetězci délky nejvýše *k*, pak se gramatika G nazývá *silná LL(k) gramatika*. Při analýze silných LL(k) gramatik jsou do zásobníku ukládány přímo symboly gramatiky a syntaktický analyzátor je řízen rozkladovou tabulkou.

Funkce FIRST a FOLLOW

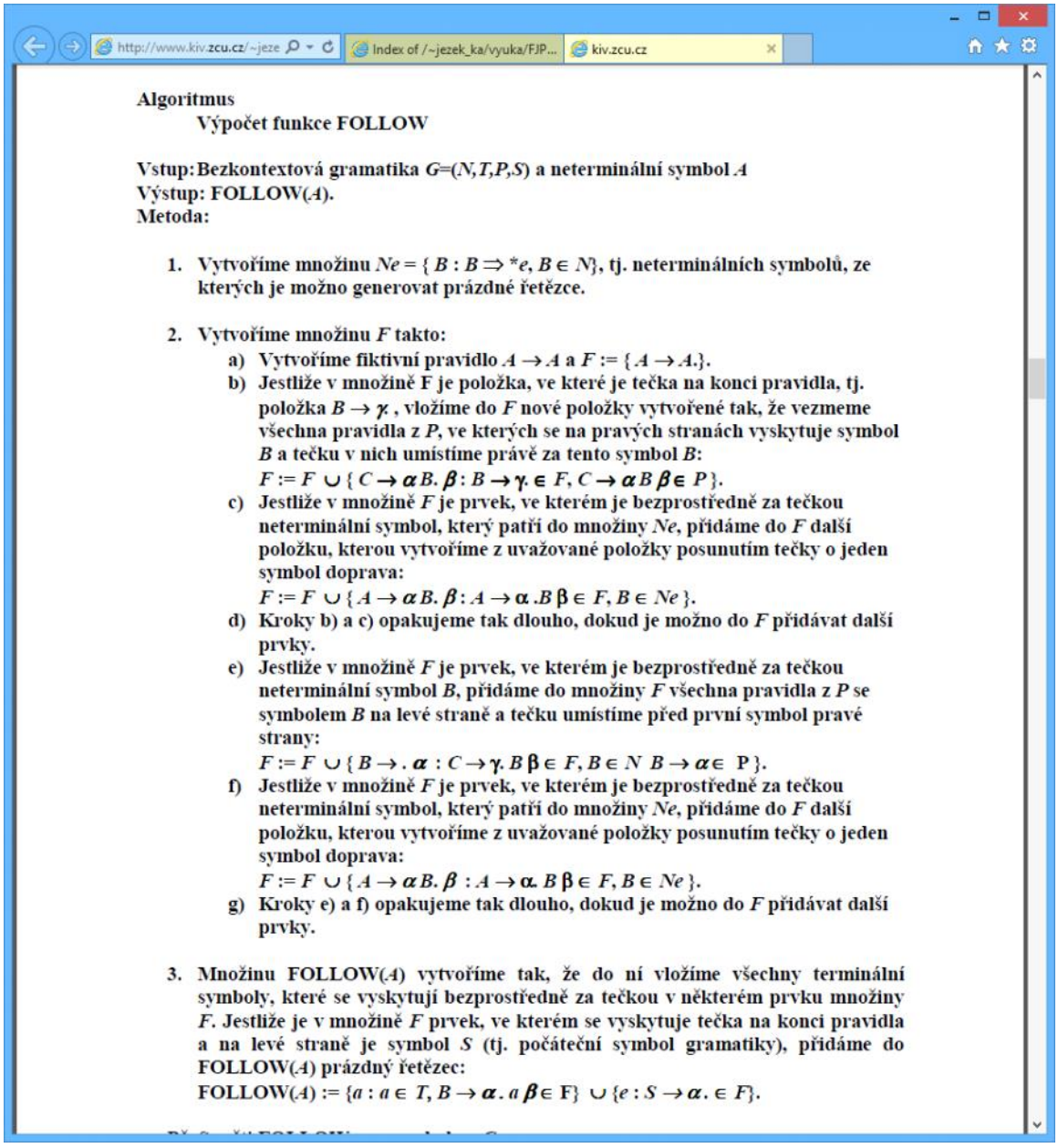
Konstrukce jak top-down, tak bottom-up parserů používá dvě funkce, FIRST a FOLLOW, spojené

s gramatikou G. Při parsování shora dolů nám FIRST a FOLLOW říkají, které přepisovací pravidlo uplatnit v závislosti na dalším vstupním symbolu. Během zotavení z chyby při panic módu mohou být množiny tokenů získané pomocí FOLLOW použity jako synchronizační tokeny.



Terminal c is in FIRST(A) and a is in FOLLOW(A)

Popis LL gramatik a LL analyzátoru



Algoritmus
Výpočet funkce FOLLOW

Vstup: Bezkontextová gramatika $G=(N,T,P,S)$ a neterminální symbol A
Výstup: FOLLOW(A).
Metoda:

1. Vytvoříme množinu $Ne = \{ B : B \Rightarrow^* e, B \in N\}$, tj. neterminálních symbolů, ze kterých je možno generovat prázdné řetězce.

2. Vytvoříme množinu F takto:
   a) Vytvoříme fiktivní pravidlo $A \rightarrow A$ a $F := \{ A \rightarrow A.\}$.
   b) Jestliže v množině F je položka, ve které je tečka na konci pravidla, tj. položka $B \rightarrow \gamma .$, vložíme do F nové položky vytvořené tak, že vezmeme všechna pravidla z P, ve kterých se na pravých stranách vyskytuje symbol B a tečku v nich umístíme právě za tento symbol B:
   $F := F \cup \{ C \rightarrow \alpha B. \beta : B \rightarrow \gamma. \in F, C \rightarrow \alpha B \beta \in P \}$.
   c) Jestliže v množině F je prvek, ve kterém je bezprostředně za tečkou neterminální symbol, který patří do množiny Ne, přidáme do F další položku, kterou vytvoříme z uvažované položky posunutím tečky o jeden symbol doprava:
   $F := F \cup \{ A \rightarrow \alpha B. \beta : A \rightarrow \alpha .B \beta \in F, B \in Ne \}$.
   d) Kroky b) a c) opakujeme tak dlouho, dokud je možno do F přidávat další prvky.
   e) Jestliže v množině F je prvek, ve kterém je bezprostředně za tečkou neterminální symbol B, přidáme do množiny F všechna pravidla z P se symbolem B na levé straně a tečku umístíme před první symbol pravé strany:
   $F := F \cup \{ B \rightarrow . \alpha : C \rightarrow \gamma. B \beta \in F, B \in N \; B \rightarrow \alpha \in P \}$.
   f) Jestliže v množině F je prvek, ve kterém je bezprostředně za tečkou neterminální symbol, který patří do množiny Ne, přidáme do F další položku, kterou vytvoříme z uvažované položky posunutím tečky o jeden symbol doprava:
   $F := F \cup \{ A \rightarrow \alpha B. \beta : A \rightarrow \alpha. B \beta \in F, B \in Ne \}$.
   g) Kroky e) a f) opakujeme tak dlouho, dokud je možno do F přidávat další prvky.

3. Množinu FOLLOW(A) vytvoříme tak, že do ní vložíme všechny terminální symboly, které se vyskytují bezprostředně za tečkou v některém prvku množiny F. Jestliže je v množině F prvek, ve kterém se vyskytuje tečka na konci pravidla a na levé straně je symbol S (tj. počáteční symbol gramatiky), přidáme do FOLLOW(A) prázdný řetězec:
   FOLLOW(A) := $\{a : a \in T, B \rightarrow \alpha . a \beta \in F\} \cup \{e : S \rightarrow \alpha . \in F\}$.

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

Řešení kolizí: http://www.kiv.zcu.cz/~lobaz/fjp/fjp11.html

Vnitřní jazyky překladačů – druhy, použití v jednotlivých fázích překladu, překlad jednoduchých jazykových konstrukcí

Thursday, May 30, 2013 8:42 AM

Po ukončení syntaktické a sémantické analýzy generují některé překladače explicitní intermediální reprezentaci zdrojového programu (mezikód). Intermediální 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. Intermediální 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. Intermediální reprezentace mohou mít různé formy.

Postfixová notace

operátory následují ihned za operandy

- A B C * D + - => A – (B * C + D)
- efektivní zpracování pomocí zásobníku, musíme vědět prioritu operátorů

Instrukce postfixoveho zapisu napr. Lit 0, A = uloz konstantu A do zasobniku, opr 0, A = proved instrukci A…

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. Na levé straně je **adresa**, na pravé **instrukce.** Adresou může být název (ze zdrojového programu, je pak nahrazen pointerem do jeho tabulky symbolů), konstanta, kompilátorem generovaná dočasná proměnná (užitečné pro optimalizaci).

Jde o linearizovanou podobu syntaktického stromu.

Překlad výrazu x+y*z na tříadresový kód:

t1 := y * z

t2 := x + t1

Další formy tříadresových instrukcí: s unárním operátorem (x = -y), *copy* instrukce (x = y), indexované copy instrukce (x = y[0]), nepodmíněný skok (goto L), podmíněný skok (if x goto L), volání procedur (call p, n; předtím uvedeno *n* parameterů).

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).

Výhoda čtveřic oproti trojicím – v optimalizaci kompilátoru, kdy jsou instrukce často přemísťovány; přesun čtveřic je ok (nová pozice se dá hned určit podle dočasných proměnných), u trojic je při posunu třeba změnit reference na výsledky, protože jsou určeny svou pozicí.

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 (jejich pozice).

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

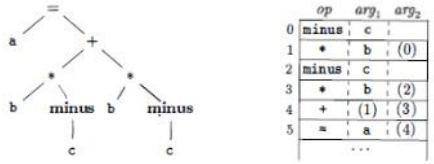
	op	arg1	arg2	res
(1)	uminus	c		t1
(2)	*	b	t1	t2
(3)	loadidx	d	b	t3
(4)	+	t2	t3	t4
(5)	:=	t4		a

Trojice

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

Překlad

u čtveřic voláme metodu pro generování instrukce a předáme jí jen parametry instrukce - např. GADD(...) = generate ADD a parametry jsou levá a pravá strana + docasna promenna;
u trojic tam musí bejt rozhodovací tabulka a kód se generuje v závislosti na tom, co je aktuálně na stacku přímo se to jmenuje Rozhodovací tabulka COMP



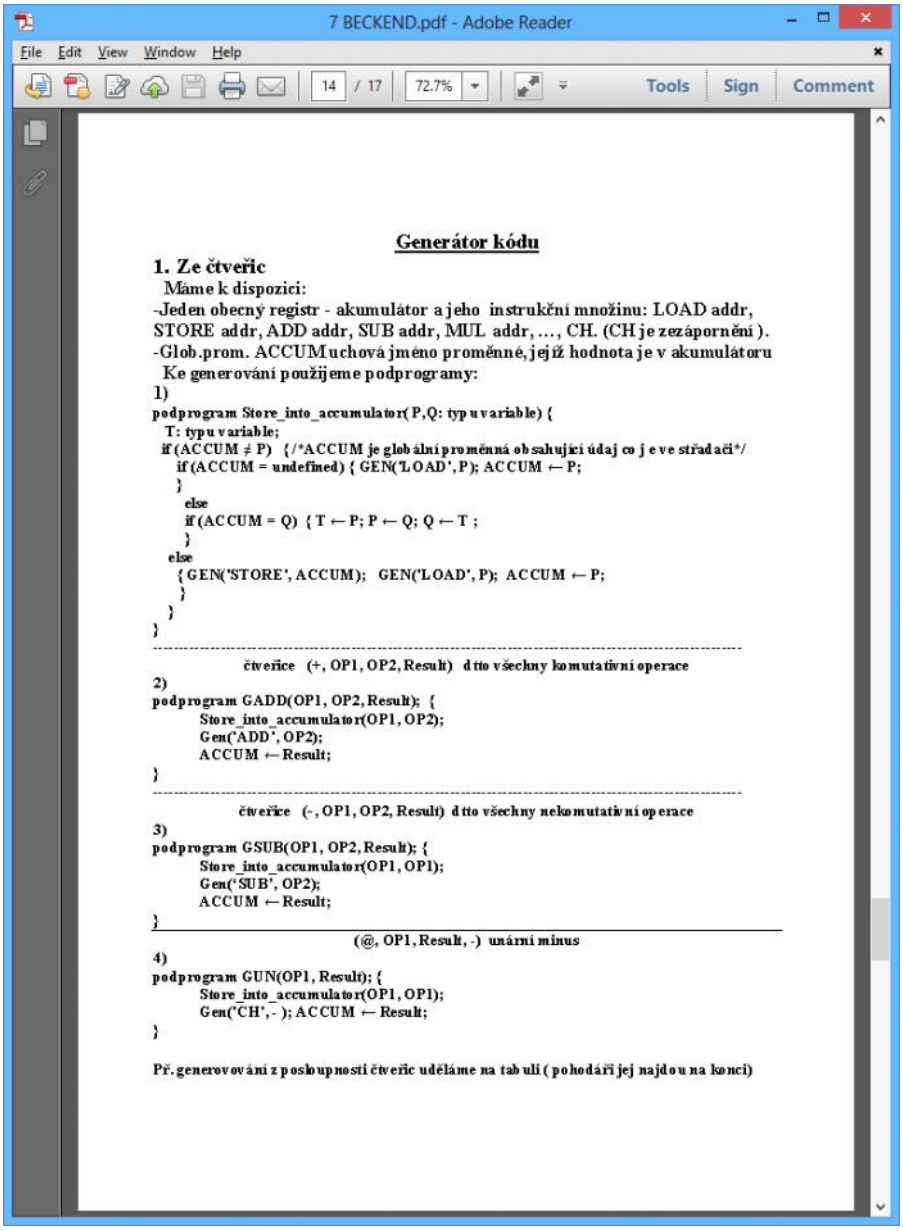
(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

(b) Triples

Figure 6.11: Representations of $a + a * (b - c) + (b - c) * d$

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



Generátor kódu

1. Ze čtveřic

Máme k dispozici:

-Jeden obecný registr - akumulátor a jeho instrukční množinu: LOAD addr, STORE addr, ADD addr, SUB addr, MUL addr, ..., CH. (CH je zezápornění).

-Glob.prom. ACCUM uchová jméno proměnné, jejíž hodnota je v akumulátoru

Ke generování použijeme podprogramy:

1)

```
podprogram Store_into_accumulator( P,Q: typu variable) {
  T: typu variable;
 if (ACCUM ≠ P)  {/*ACCUM je globální proměnná obsahující údaj co je ve střadači*/
    if (ACCUM = undefined) { GEN('LOAD', P); ACCUM ← P;
    }
      else
      if (ACCUM = Q) { T ← P; P ← Q; Q ← T ;
      }
   else
    { GEN('STORE', ACCUM);  GEN('LOAD', P);  ACCUM ← P;
    }
  }
}
```

čtveřice (+, OP1, OP2, Result) dtto všechny komutativní operace

2)

```
podprogram GADD(OP1, OP2, Result); {
        Store_into_accumulator(OP1, OP2);
        Gen('ADD', OP2);
        ACCUM ← Result;
}
```

čtveřice (-, OP1, OP2, Result) dtto všechny nekomutativní operace

3)

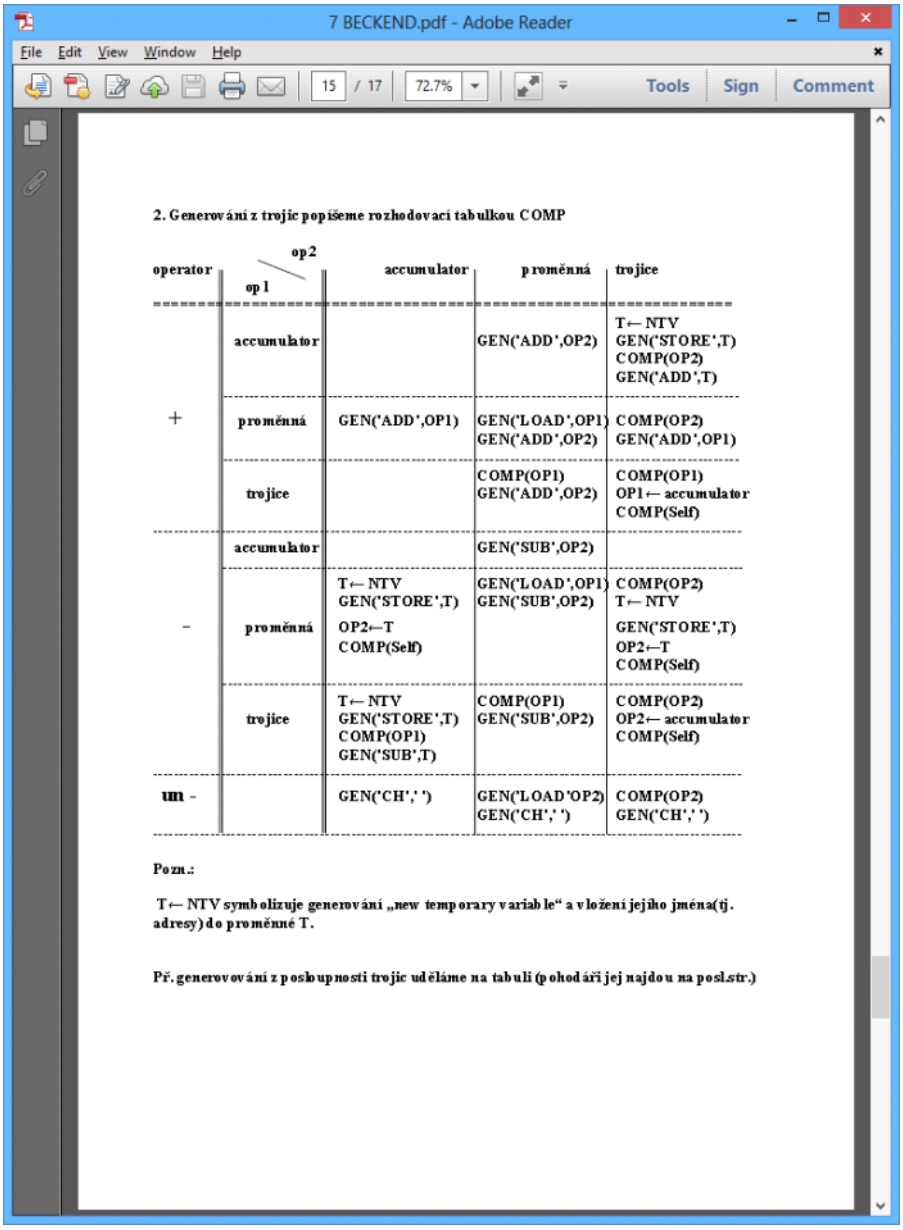
```
podprogram GSUB(OP1, OP2, Result); {
        Store_into_accumulator(OP1, OP1);
        Gen('SUB', OP2);
        ACCUM ← Result;
}
```

(@, OP1, Result, -) unární minus

4)

```
podprogram GUN(OP1, Result); {
        Store_into_accumulator(OP1, OP1);
        Gen('CH',- ); ACCUM ← Result;
}
```

Př. generovování z posloupnosti čtveřic uděláme na tabuli (pohodářiji najdou na konci)



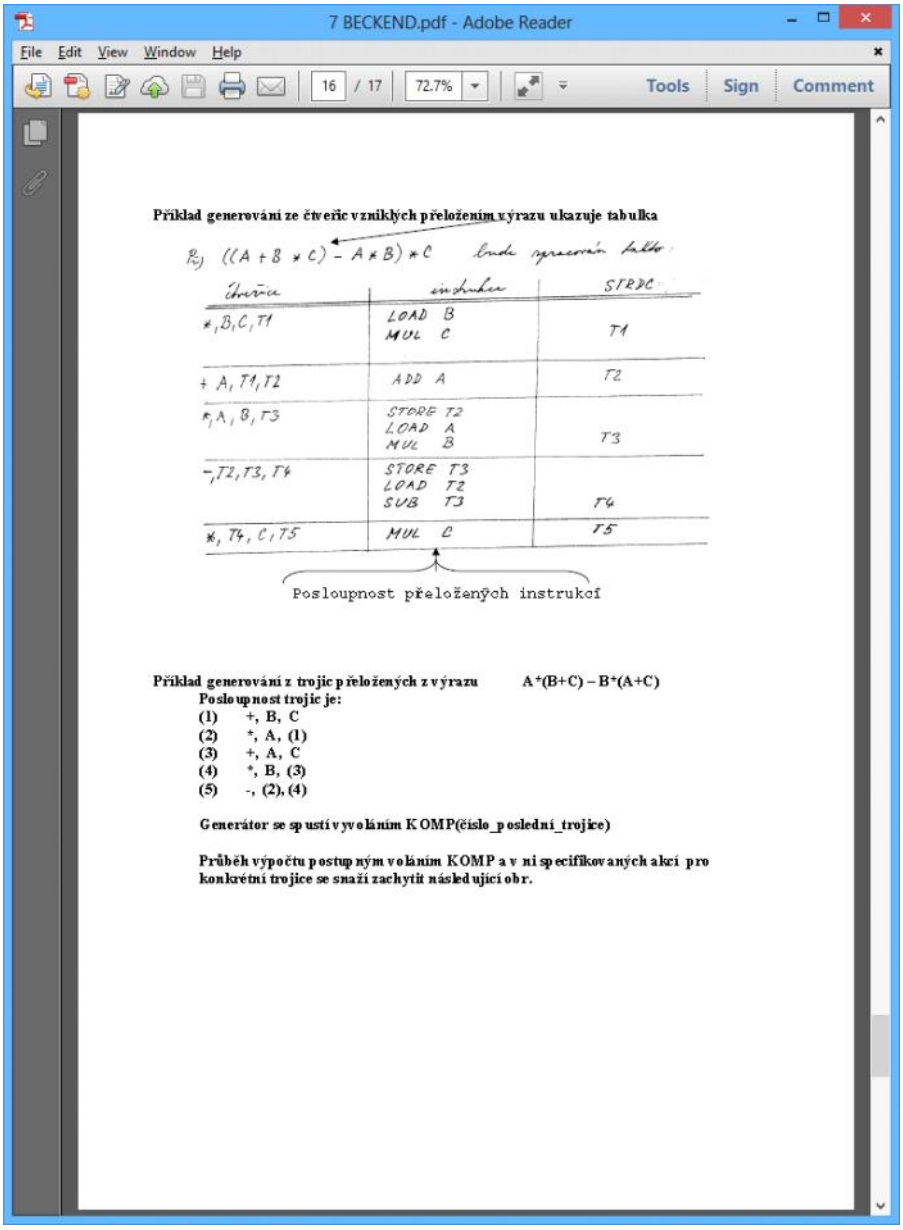
2. Generování z trojic popíšeme rozhodovací tabulkou COMP

operator	op1 \ op2	accumulator	proměnná	trojice
+	accumulator		GEN('ADD',OP2)	T← NTV GEN('STORE',T) COMP(OP2) GEN('ADD',T)
	proměnná	GEN('ADD',OP1)	GEN('LOAD',OP1) GEN('ADD',OP2)	COMP(OP2) GEN('ADD',OP1)
	trojice		COMP(OP1) GEN('ADD',OP2)	COMP(OP1) OP1← accumulator COMP(Self)
-	accumulator		GEN('SUB',OP2)	
	proměnná	T← NTV GEN('STORE',T) OP2←T COMP(Self)	GEN('LOAD',OP1) GEN('SUB',OP2)	COMP(OP2) T← NTV GEN('STORE',T) OP2←T COMP(Self)
	trojice	T← NTV GEN('STORE',T) COMP(OP1) GEN('SUB',T)	COMP(OP1) GEN('SUB',OP2)	COMP(OP2) OP2← accumulator COMP(Self)
un -		GEN('CH',' ')	GEN('LOAD'OP2) GEN('CH',' ')	COMP(OP2) GEN('CH',' ')

Pozn.:

T← NTV symbolizuje generování „new temporary variable" a vložení jejího jména(tj. adresy) do proměnné T.

Př. generovování z posloupnosti trojic uděláme na tabuli (pohodáří jej najdou na posl.str.)



Příklad generování z trojic přeložených z výrazu $A*(B+C) - B*(A+C)$

Posloupnost trojic je:

(1) +, B, C
(2) *, A, (1)
(3) +, A, C
(4) *, B, (3)
(5) -, (2), (4)

Generátor se spustí vyvoláním KOMP(číslo_poslední_trojice)

Průběh výpočtu postupným voláním KOMP a v ní specifikovaných akcí pro konkrétní trojice se snaží zachytit následující obr.

Tabulka symbolů – obsah, způsob manipulace při vytváření a využívání při překladu

Thursday, May 30, 2013 8:42 AM

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.

OBSAH

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 odlišila 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.

ZPŮSOB MANIPULACE PŘI VYTVÁŘENÍ A VYUŽÍVÁNÍ PŘI PŘEKLADU

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.

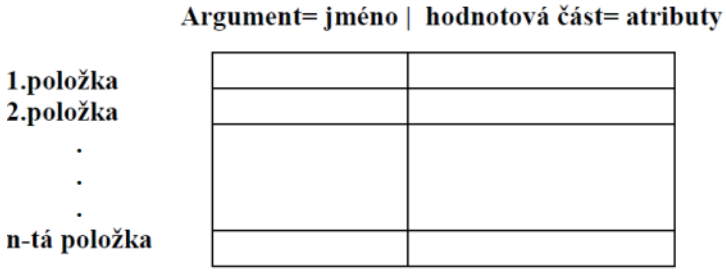
Speciální implementaci vyžaduje tabulka symbolů **pro jazyk s blokovou strukturou**, jako je třeba Pascal. Rozlišují se zde **lokální a globální objekty** a přístupnost lokálních je omezena. Každá proměnná je viditelná v tom bloku, ve kterém je deklarovaná, a také ve všech blocích vnořených. Když v určitém 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.

Tabulka symbolů

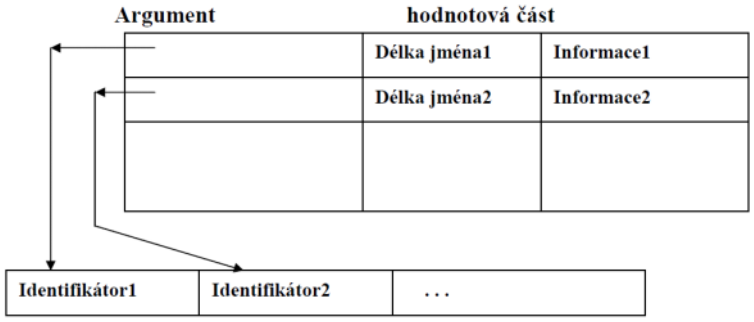
- uchovává informace o objektech
- umožňuje kontextové kontroly
- umožňuje operace
    a. inicializaci informace pro standardní jména
    b. vyhledání jména
    c. doplnění informace ke jménu
    d. přidání položky pro nové jméno
    e. vypuštění položky či skupiny položek

Struktura tabulky symbolů

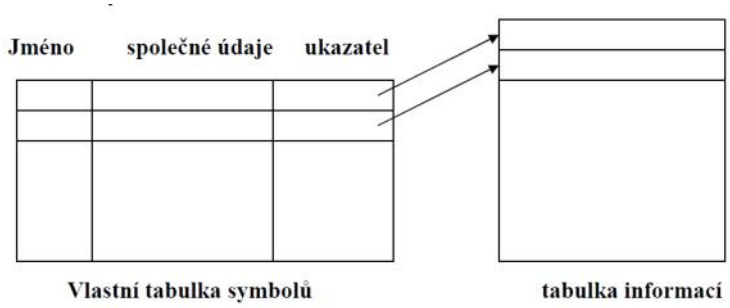
- s jednoduchou strukturou



- s oddělenou tabulkou identifikátorů

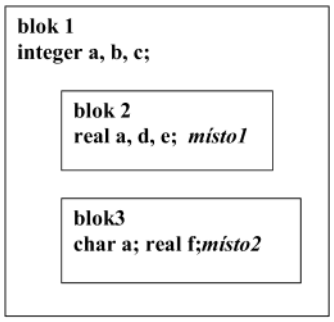


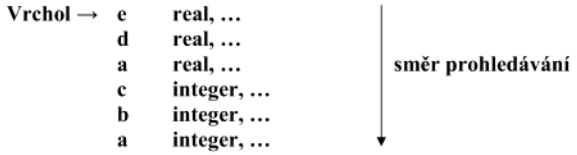
- s oddělenou tabulkou informací

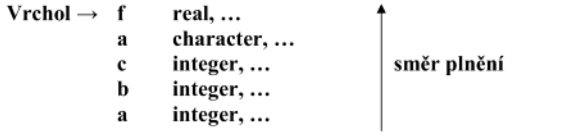


- uspořádané do podoby zásobníku

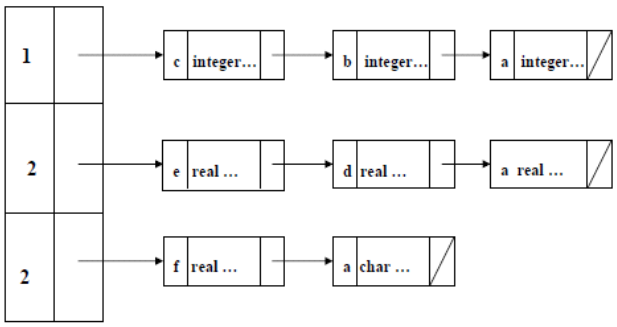
Tabulka symbolů uspořádaná do podoby zásobníku (pro jazyky s blokovou strukturou).
Rozsahová jednotka je blok, modul, funkce, balík,…
Respektuje zásady lokality



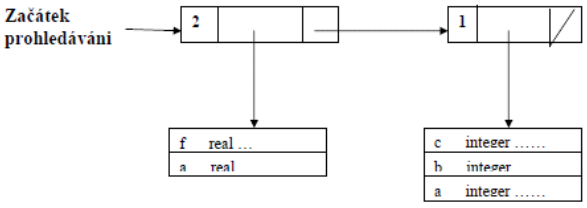




- s blokovou strukturou



Překládá-li se uvnitř bloku 3:



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.

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

Principy přidělování paměti překladačem

Thursday, May 30, 2013 8:43 AM

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

Základní způsoby přidělování:

- **Statické** (přidělení paměti v čase překladu)
- **Dynamické** (přiděleno v run time) **- v zásobíku** nebo **na haldě**

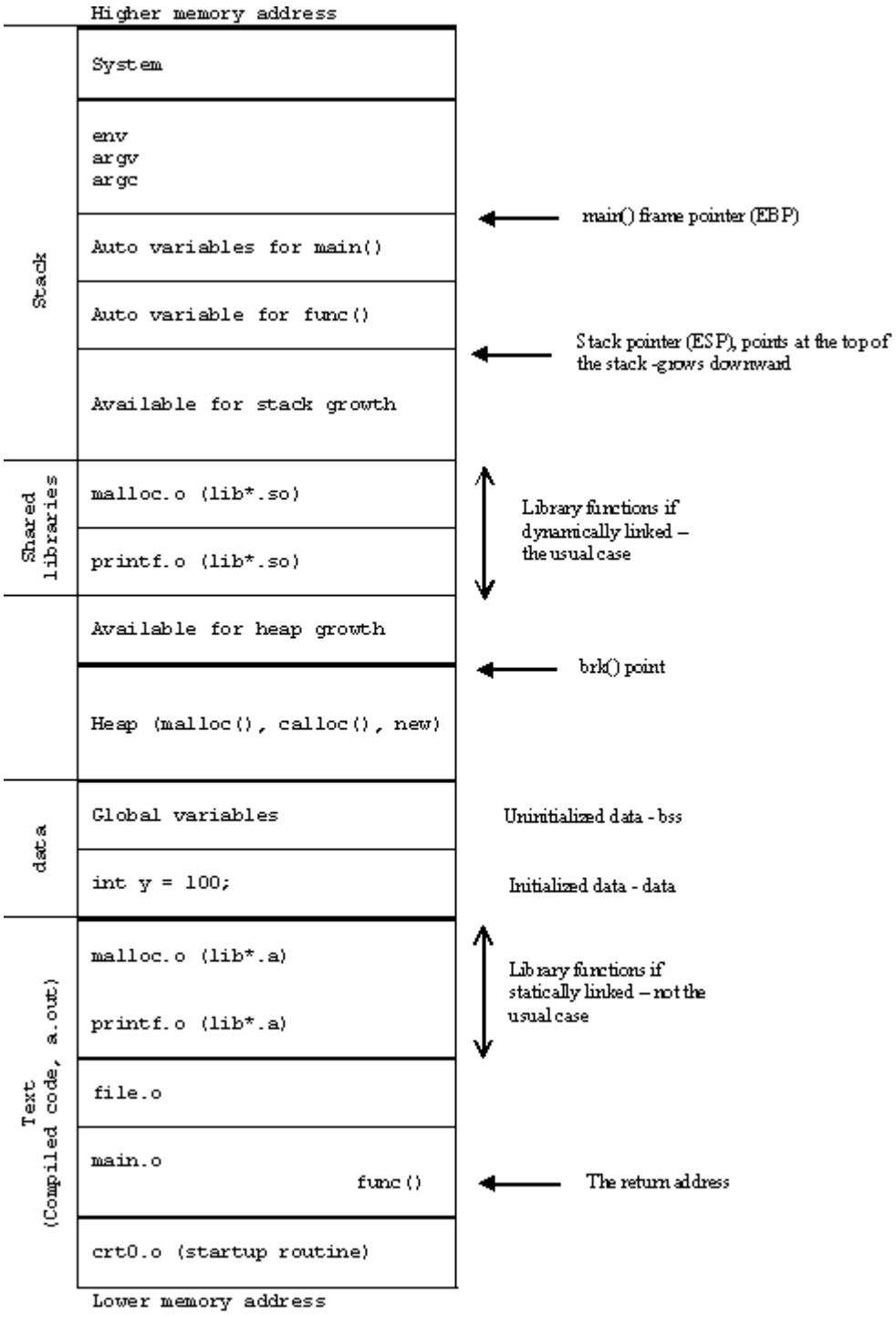
Velikost vygenerovaného kódu je známa již v době překladu, takže jej může překladač umístit **do staticky definované oblasti** (*Code/cílový kód programu*), obvykle na začátek přiděleného paměťového prostoru.

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).

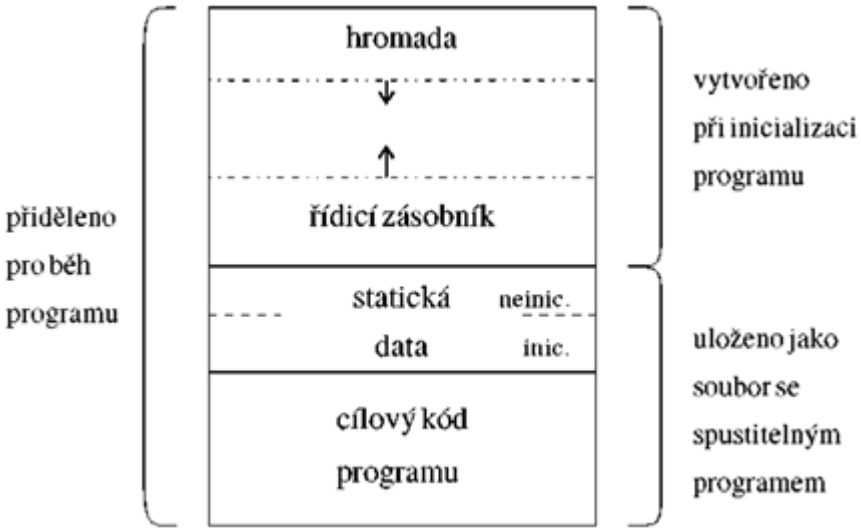
Jazyky umožňující rekurzi (Pascal, C, …) využívají pro aktivace podprogramů řídicího **zásobníku (stack)**, do kterého se ukládají jednotlivé ***aktivační záznamy*** (AZ jsou generovány při voláních procedur).

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 zvaná **hromada (heap)**.

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. **Zásobník roste směrem k nižším adresám, hromada směrem k vyšším**. Nedostatek paměti se rozpozná tehdy, jestliže ukazatel konce některé oblasti překročí hodnotu ukazatele konce druhé oblasti.



From <http://www.tenouk.com/ModuleW_files/ccompilerlinker006.png>



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

- **Statické** přidělování paměti v době překladu
- **Dynamické** přidělování paměti za běhu programu:
  - Přidělování paměti na *zásobníku*
  - Přidělování paměti z *hromady*

Statické přidělování paměti v době překladu

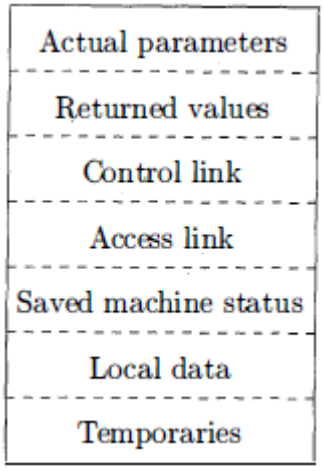
Při statickém přidělování paměti jsou všem objektům v programu **přiděleny adresy již v době překladu (globální data)**. Při kterémkoliv volání podprogramu jsou jeho lokální proměnné vždy na stejném místě, což umožňuje zachovávat hodnoty lokálních proměnných nezměněné mezi různými aktivacemi podprogramu. 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**.

Aktivace procedur při běhu programu jde znázornit ***aktivačním stromem***. Co uzel, to jedna aktivace, kořen je aktivací hlavní procedury, která je volána po spuštění programu; potomci uzlu *p* = volání procedur z procedury *p*. Kořen aktivačního stromu je na dně zásobníku, poslední aktivace má svůj záznam na vrcholu zásobníku.

Každá „živá“ aktivace má **aktivační záznam**. Obsah aktivačního záznamu se liší podle implementovaného jazyka.



1. ***dočasné hodnoty*** – vypadnou po vyhodnocení výrazů, jsou tu, pokud nemohou být udržovány v registrech
2. ***lokální data*** – patřící k dané proceduře s příslušným aktivačním záznamem
3. ***uložený strojový status*** – info o stavu stroje před voláním procedury. Typicky jde o:
   - *návratovou adresu* (= hodnota program counteru, kam se má pak procedura vrátit) a o
   - *obsah registrů* použitých procedurou (musí být obnoveny po návratu z procedury)
4. ***access link = statický ukazatel*** – pro lokaci dat, která procedura potřebuje, ale která se nachází v jiném aktivačním záznamu
5. ***control link*** – ukazuje na aktivační záznam volajícího (caller)
6. ***vrácené hodnoty*** – prostor pro návratovou hodnotu volané funkce (kvůli rychlosti lepší dávat do registru)
7. ***vlastní parametry*** – parametry použité volající procedurou; pokud je to možné, jsou umístěny radši v registrech kvůli výkonnosti.

Při implementaci přidělování paměti na zásobníku bývá **jeden registr vyhrazen jako ukazatel na začátek aktivačního záznamu na vrcholu zásobníku**. Relativně 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.

přístup k nelokálním proměnným při statickém =lexikálním rozsahu platnosti jmen. To řeší tzv. **řetězec statických ukazatelů** (*access links*). Pro zrychlení přístupu k nelokálním proměnným se zavádí vektor ukazatelů – **displej**. Zamezí se tak průchod aktivačními záznamy pro hluboko zanořené podprogramy.

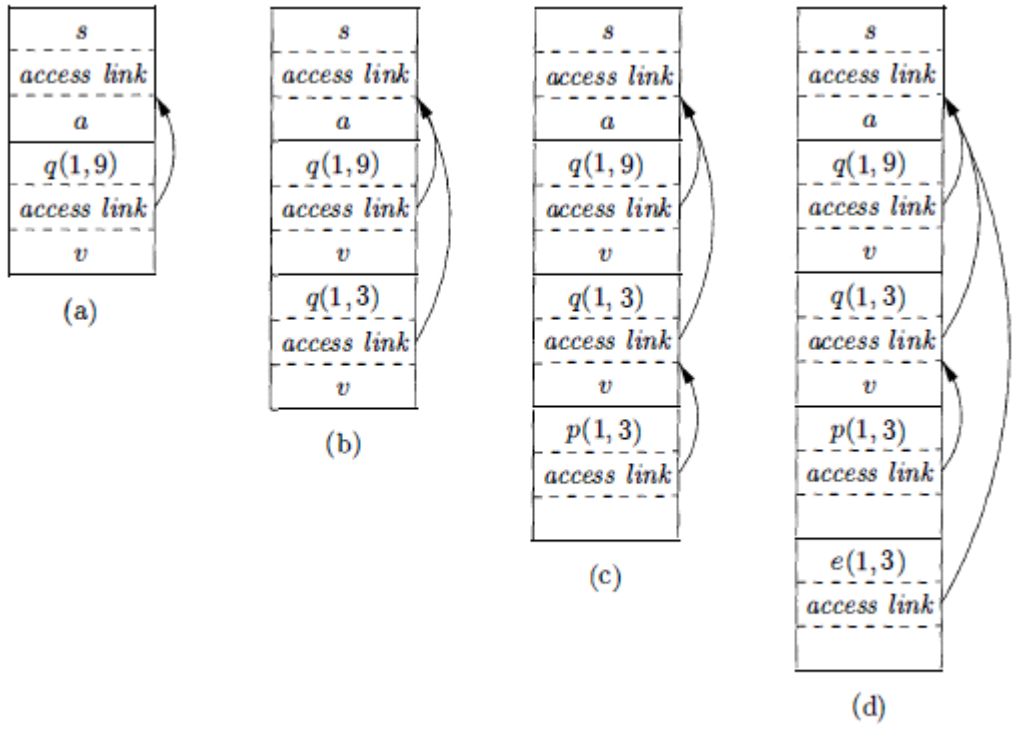


Figure 7.11: Access links for finding nonlocal data

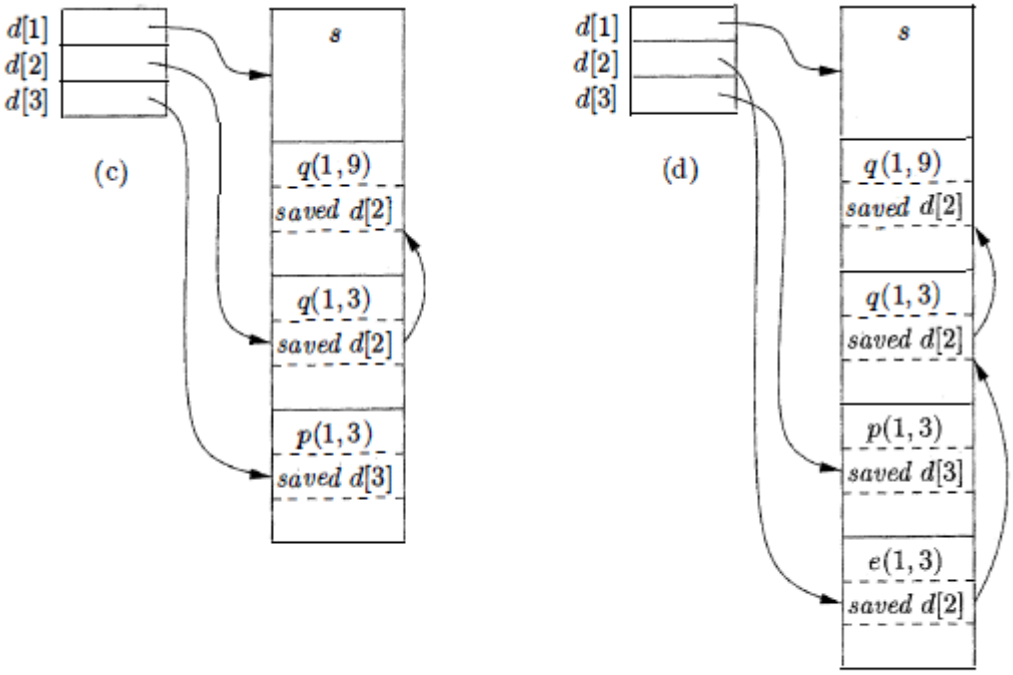


Figure 7.14: Maintaining the display

Kompilátor musí v čase kompilace určit rozvržení aktivačních záznamů a vygenerovat kód, který korektně přistupuje na místa v aktivačním záznamu. Proto *musí* být rozvržení AZ a generátor kódu navrhovány společně.

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.

Správce paměti (memory manager) alokuje a dealokuje místo na heapu. V důsledku toho může dojít k fragmentaci heapu (vznik malých, nesouvislých míst = *děr*). Strategie *best fit* = alokuj nejmenší vhodnou a dostupnou díru.

Garbage collection hledá místo na heapu, které se už nepoužívá a může být proto realokované pro uchovávání dalších dat (Java, C#). Automaticky uvolňuje již nepoužívané objekty z paměti.

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

Vlastnosti jazykových konstrukcí pro statický a pro dynamický způsob přidělování paměti

Thursday, May 30, 2013 8:43 AM

Přímo podle Ježka:

Pro jazykovou konstrukci, která umožňuje jen statické přidělování, je možné řešit alokovanou paměť už během překladu. U těch dynamických není předem známo, kolik paměti je třeba vyhradit, je nutné to provádět až během vykonávání programu.

Jsou rekurze (nevíme předem, kolikrát se zanoříme)
Dynamické proměnné (new)
Dynamické typy (pole, jejichž velikost je daná výrazem)

Řeknu si, že chci udělat vlastní programovací jazyk a chci vědět, jak ho mám navrhnout - je třeba si uvědomit, co od něj budu potřebovat. Pokud potřebuju např. rekurzi, musím ho navrhnout tak, aby podporoval dynamické typy.

U statického přidělování paměti si mohu dovolit dělat statické proměnné, metody bez rekurze apod.
U dynamického pak vše ostatní jako rekurzi apod.

http://service.felk.cvut.cz/courses/X36PJP/Skripta_prednasky.pdf !!!

Důležitá hlediska jazykových konstrukcí:

- Dynamické typy
- Dynamické proměnné
- Rekurze
- Konstrukce pro paralelní výpočty

Podstatný je rovněž způsob:

- Omezování existence entit v programu
- Předávání parametrů funkce (hodnotou, odkazem)
- Určování přístupu k nelokálním entitám
    - na základě statického vnořování rozsahových jednotek,
    - na základě dynamického vnoření rozsahových jednotek.

Statické přidělování paměti:

- Globální proměnné
- Statické proměnné
- Proměnné jazyka bez rekurze (i s blokovou strukturou) (možno staticky na zásobníku)

Dynamické přidělování paměti (na hromadě):

- Dynamické typy a proměnné (překladač neví v době překladu kolik jich bude potřebovat)
- Proměnné předávané odkazem
- Pointery v C pro dynamicky alokované proměnné

Předávání parametrů podprogramům

- hodnotou (C, C++, Java, C#) formální parametr podprogramu je lokální proměnnou (tohoto podprogramu) do níž se předá hodnota (proměnná je zkopírována do zásobníku podprogramu)
- odkazem (C, C++ je-li parametrem pointer, objektové parametry Javy, C# parametry označené ref) předá informaci o umístění skutečného parametru (předána adresa na proměnnou) - u Javy se všechno předává prostřednictvím hodnoty (pass-by-value) tj. v případě instancí tříd (objektů) dochází k předávání **adresy** objektu ???
- výsledkem - formální parametr je lokální proměnnou z níž se předá hodnota do skutečného parametru před návratem z podprogramu

Dynamické přidělování v zásobníku

Aktivační záznam obsahuje místo pro:

- Lokální proměnné
- Parametry
- Návratovou adresu
- Funkční hodnotu (je-li podpr. funkcí)
- Pomocné proměnné (pro mezivýsledky)
- Další informace potřebné k uspořádání aktivačních záznamů

Statická typová kontrola – referenční prostředí podprogramů je definováno staticky, tj. při překladu zdrojového programu. Pro každou deklaraci je staticky vymezen rozsah platnosti, tj. část zdrojového kódu, ve kterém lze deklarované jméno použít. V podprogramu pak kromě lokálních jmen lze použít ta nelokání jména, do jejichž rozsahu platnosti je definice podprogramu vnořena. Statické referenční prostředí je často založeno na blokové struktuře programu.

Statické referenční prostředí je při překladu reprezentováno tabulkou symbolů. Překlad deklarace znamená rozšíření tabulky symbolů o nový záznam, při dosažení místa konce platnosti deklarace se záznam odstraní nebo skryje. Při překladu těla podprogramu překladač na základě tabulky symbolů pro každé jméno rozhodne, zda je či není v daném místě použitelné a jaký datový objekt (nebo jiný programový prvek) označuje. Tím se dosáhne vyšší bezpečnosti programu (nepoužitelné jméno je odhaleno již při překladu) i vyšší efektivity cílového programu.

Dynamická typová kontrola – např. Lisp, referenční prostředí podprogramů je definováno dynamicky. Dynamicky definované referenční prostředí **se nevytváří ani nekontroluje při překladu, ale až při provádění programu**. Při spuštění programu se vytvoří referenční prostředí tvořené vazbami jmen definovaných jazykem. Při každém vstupu do podprogramu se referenční prostředí rozšíří o vazby lokálních jmen podprogramu, při návratu z podprogramu se jeho lokální prostředí odstraní. Při provádění příkazů se pro každé jméno hledá jeho vazba.

Dyn. Definované ref. Prostředí snižuje bezpečnost programu i jeho efektivitu (význam jména se hledá při provádění programu). Jeho výhodou je jednoduchá sémantika – nenajde-li se jméno v lokálním prostředí podprogramu A, hledá se v lokálním prostředí podprogramu B, ze kterého byl podprogram A vyvolán, případně v lokálním prostředí podprogramu C, z čehož byl vyvolán podprogram B apod.

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

Printout

Monday, June 3, 2013 1:08 PM

Metody přidělování paměti

Základní způsoby: -Statické (přidělění paměti v čase překladu)
-Dynamické (přiděleno v run time) — v zásobníku / na haldě

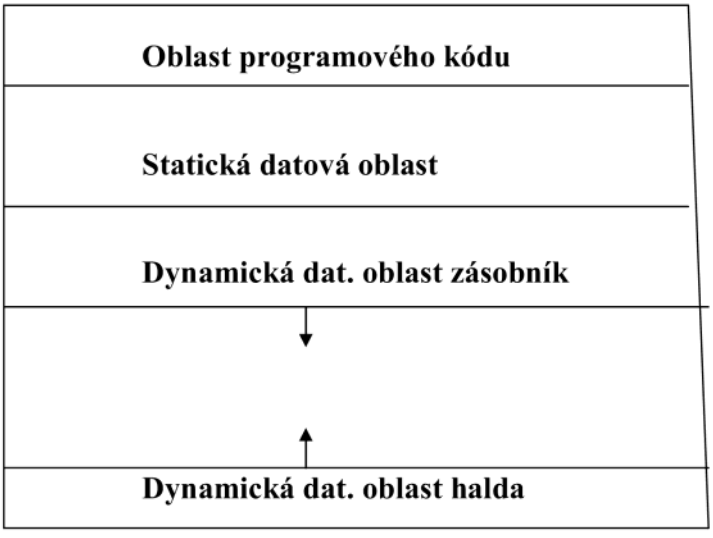
Důležitá hlediska jazykových konstrukcí:

- Dynamické typy
- Dynamické proměnné
- Rekurze
- Konstrukce pro paralelní výpočty

Podstatný je rovněž způsob:

- Omezování existence entit v programu (namespace, package, blok…)
- Určování přístupu k nelokálním entitám
  - na základě statického vnořování rozsahových jednotek,
  - na základě dynamického vnoření rozsahových jednotek.

Rozdělění paměti cílového programu



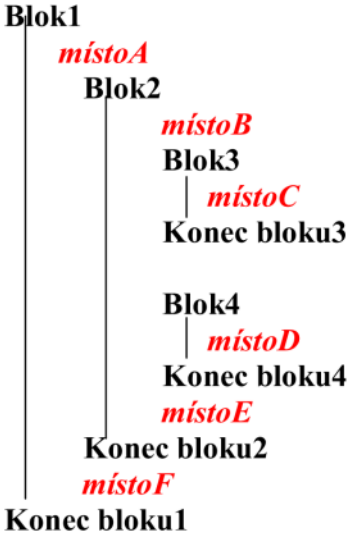
Page 2

Monday, June 3, 2013 1:08 PM

Statické přidělování paměti lze použít pro:

- **Globální proměnné**
- **Static proměnné**
- **Proměnné jazyka bez rekurze (i s vnořenou blokovou strukturou)**

Př.



Statické přidělování lze realizovat pomocí zásobníku

Ukažme obsah zásobníku v různých okamžicích výpočtu

Bloky

		3	4		
	2	2	2	2	
1	1	1	1	1	1
Místo A	**B**	**C**	**D**	**E**	**F**

Vnořování podprogramů (funkcí, metod) je ale složitější, viz dále.

Page 3

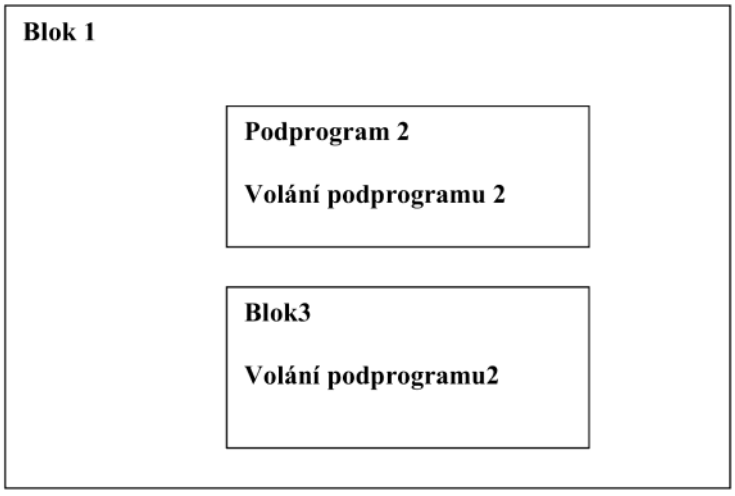
Monday, June 3, 2013 1:08 PM

Dynamické přidělování v zásobníku

Část paměti přidělovaná při vstupu výpočtu do rozsahové jednotky programu se nazývá Aktivační Záznam (AZ představuje lokální prostředí výpočtu). Obsahuje místo pro:

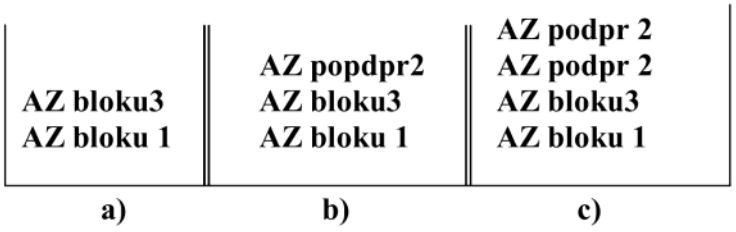
- **Lokální proměnné**
- **Parametry (je-li rozsahovou jednotkou podprogram či funkce)**
- **Návratovou adresu („ „)**
- **Funkční hodnotu (je-li rozsahová jednotka funkcí)**
- **Pomocné proměnné pro mezivýsledky (také možno v registrech)**
- **Další informace potřebné k uspořádání aktivačních záznamů**

Př.1



? stav výpočtového zásobníku v různých časech výpočtu

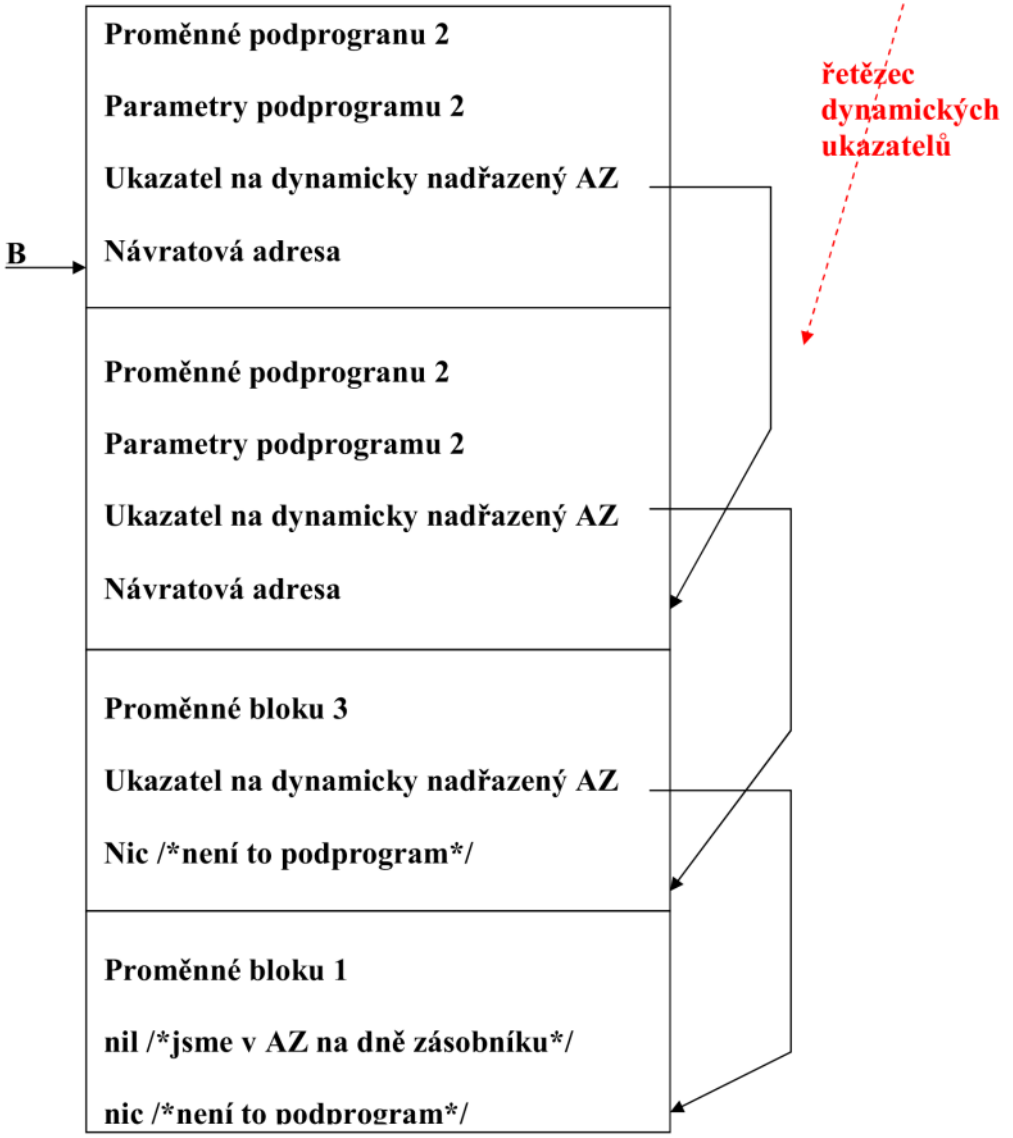
a) **Při vstupu do bloku 3**
b) **Při prvním volání podprogramu 2**
c) **Při rekurzivním vyvolání podprogramu 2**



Page 4

Monday, June 3, 2013 1:08 PM

Jaké bude uspořádání aktivačních záznamů při rekurz. vyvolání podpr.2 ?
Pro rušení AZ při výstupu z jednotky potřebujeme tzv dynamický ukazatel



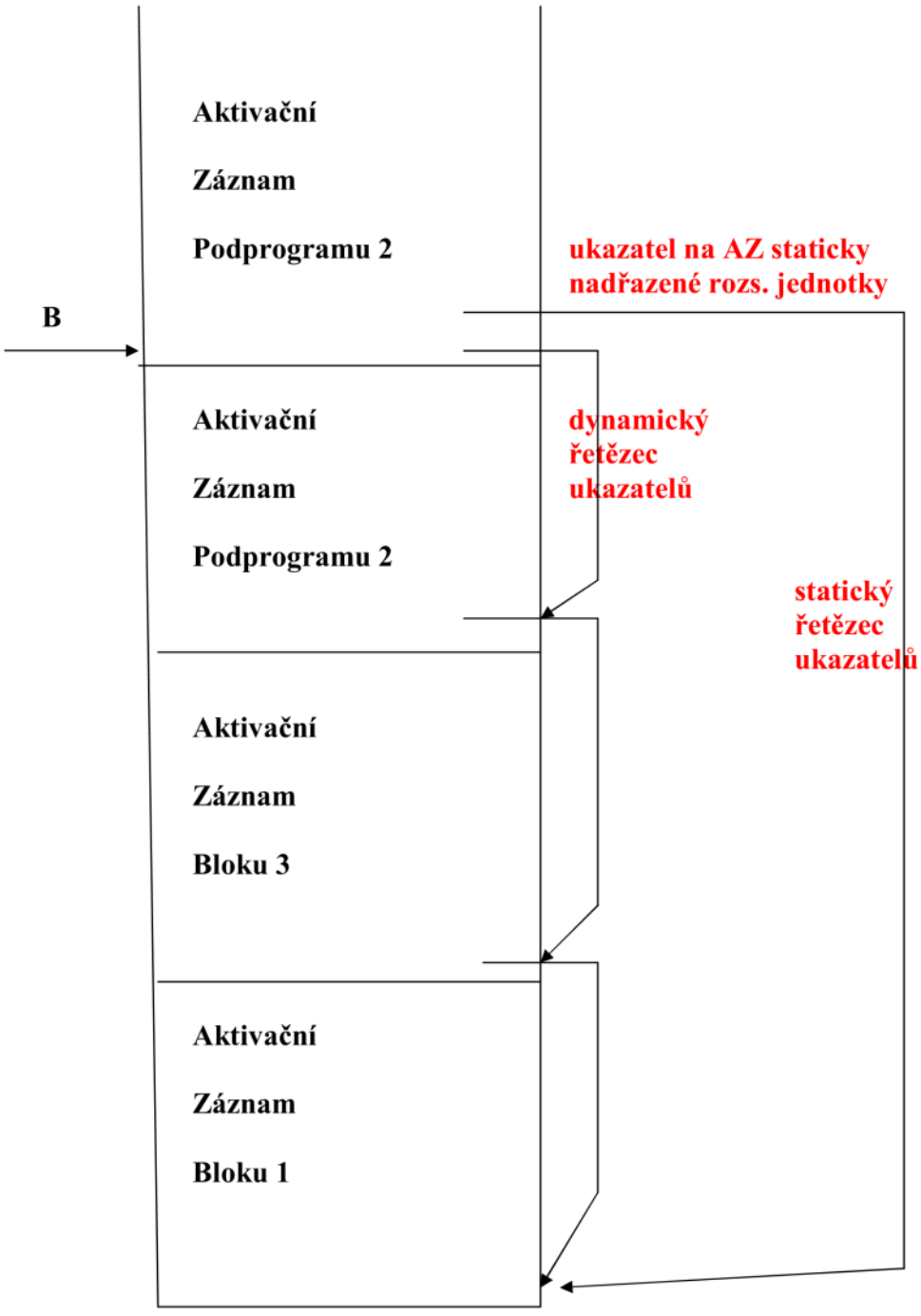
Obr. Zásobník při rekurzivním volání podprogramu 2

B je registr ukazující na vrcholový AZ

Potřebujeme ještě vyřešit přístup k nelokálním proměnným při statickém = lexikálním rozsahu platnosti jmen. To řeší tzv. řetězec statických ukazatelů

Page 5

Monday, June 3, 2013 1:08 PM



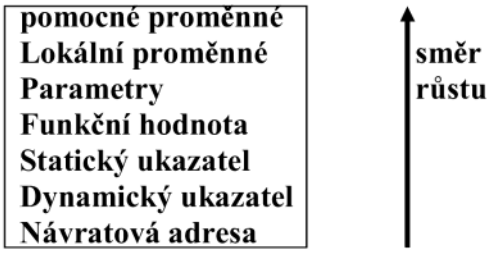
Obr. Zásobník se statickým (ukazuje na lexikálně nadřazený AZ) a dynamickým řetězcem ukazatelů při rekurzivním volání podprogramu 2
Pozn.: Statický uk. je nakreslen (pro přehlednost) jen u AZ rek. volání podpr.2

Page 6

Monday, June 3, 2013 1:08 PM

Vytváření řetězců ukazatelů

Nechť AZ má tvar:



Uvažujme zásobník Z, s vrcholem (nejvyšší zabranou adresou) T,
n je hladina deklarace , m je hladina volání
Při vstupu do rozsahové jednotky (vyvolání podprogramu nebo vstupu výpočtu do bloku = Aktivace rozsahové jednotky):

A1) Z[T + 1] ← návratová adresa /* pouze u podprogramů*/
A2) Z[T + 2] ← B /*nastavení dynamického ukazatele*/
A3) Z[T + 3] ← B
For i ← 1 to m – n do Z[T + 3] ← Z[Z[T + 3] + 2] /*nastavení statického ukazatele*/
A4) B ← T + 1 /*nastavení bázového registru*/
A5) T ← T + velikost aktivačního záznamu
A6) skok na první instrukci podprogramu a uložení do Z údajů o skutečných parametrech /*pouze u podprogramů*/

Pozn. Je-li podprogram překládán odděleně (neznámá velikost jeho AZ), pak je úprava T provedena až na začátku volaného podprogramu.

Při výstupu z rozsahové jednotky (Návrat z podprogramu nebo průchod koncem bloku):

N1) T ← B – 1
N2) B ← Z[B + 1]
N3) skok na adresu uloženou v Z[T + 1] /*pouze u podprogramů*/

Výstup z rozsahové jednotky nelokálním skokem (hladina n deklarace návěští je menší než hladina m místa s příkazem skoku)
Vždy platí n ≤ m

```
S1)  for i ← 1 to m – n do { Pom ← B
                             repeat     T ← B – 1
                                        B ← Z[ B + 1]
                             until  B ≠ Z[ POM + 2]
                           }
```

S2) skok na adresu, kterou návěští představuje

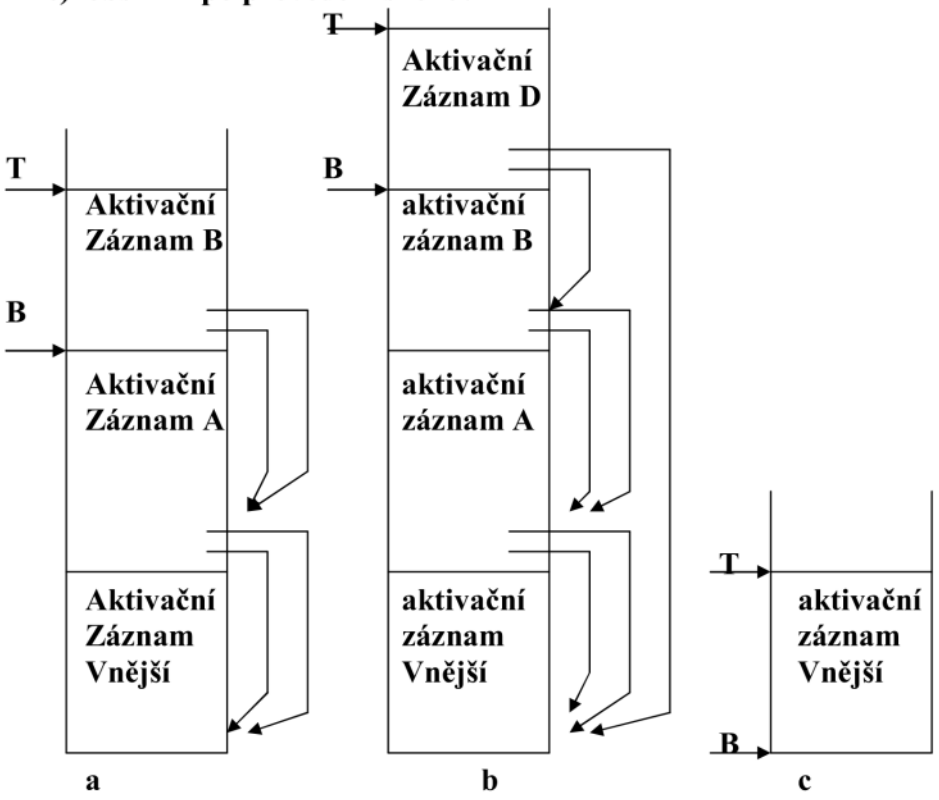
Page 7

Monday, June 3, 2013 1:08 PM

Př.2

```
Podprogram Vnější //označme n hladinu deklarace
Label 1                    // a m hladinu místa použití
        Podprogram A
                Podprogram B
                Volání D  //zde  n=1; m=3
        Volání B
        Podprogram D
        Goto 1        //zde n=1; m=2
Volání A
1:
```

a) obsah Z při provádění B, před voláním D,
b) obsah Z po vyvolání D, ped provedením nelokálního skoku,
c) obsah Z po provedení skoku.

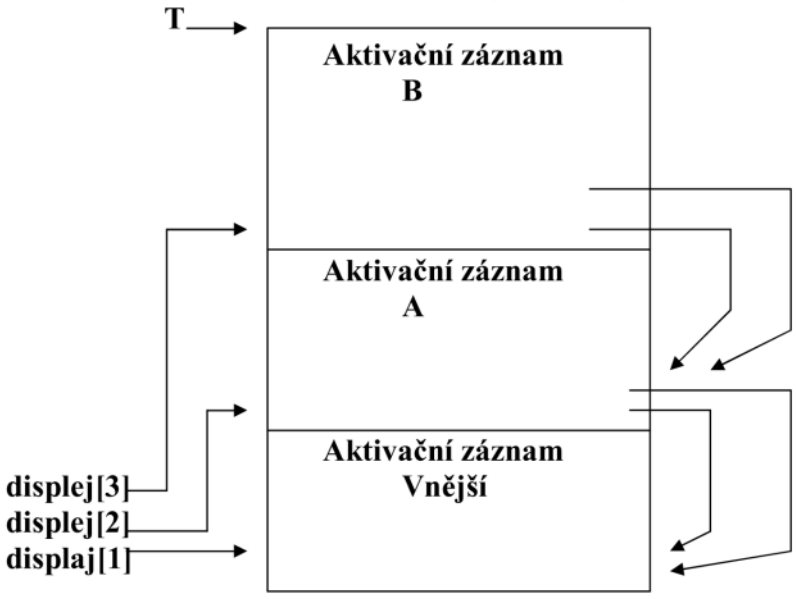


ad b) je to stav v okamžiku volání podpr. s hladinou deklarace 1, volaného v místě s hladinou 3
ad c) stav po výskoku z hladiny 2 do místa s hladinou 1

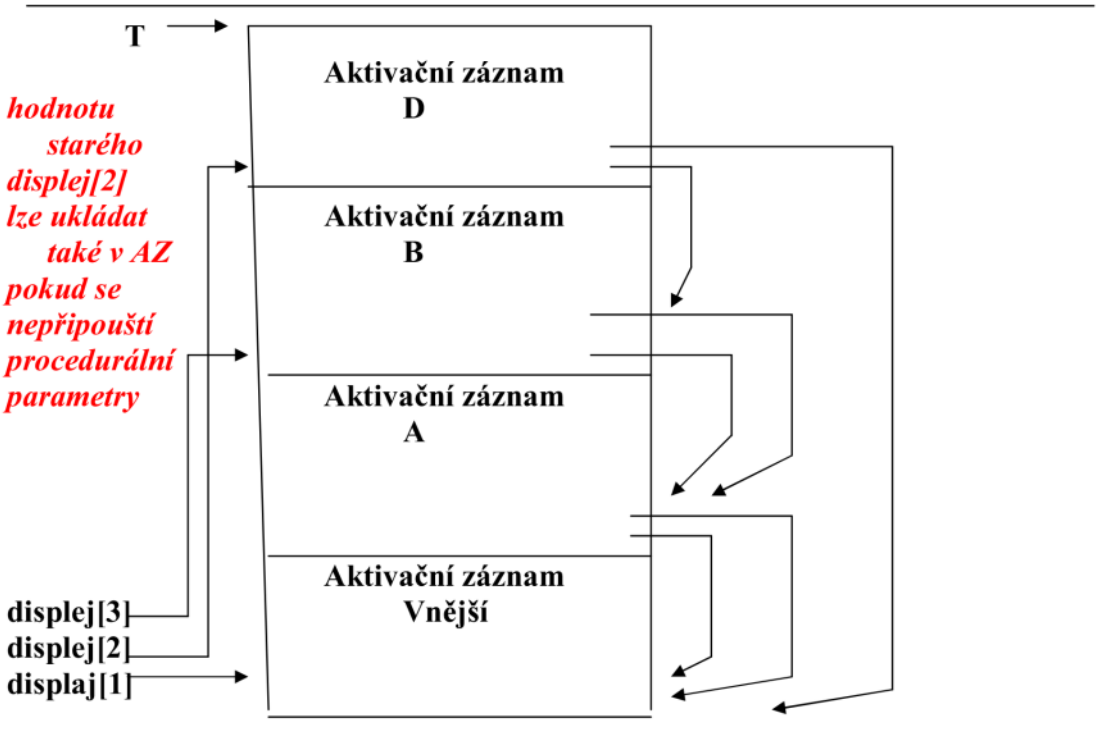
Page 8

Monday, June 3, 2013 1:08 PM

Zrychlení přístupu k nelokálním proměnným
(pomocí vektoru ukazatelů displej[i], kde i je hladina rozs. jedn.)



Obr. Stav Z při výpočtu B z př.2



Obr. Stav Z při výpočtu D z př.2

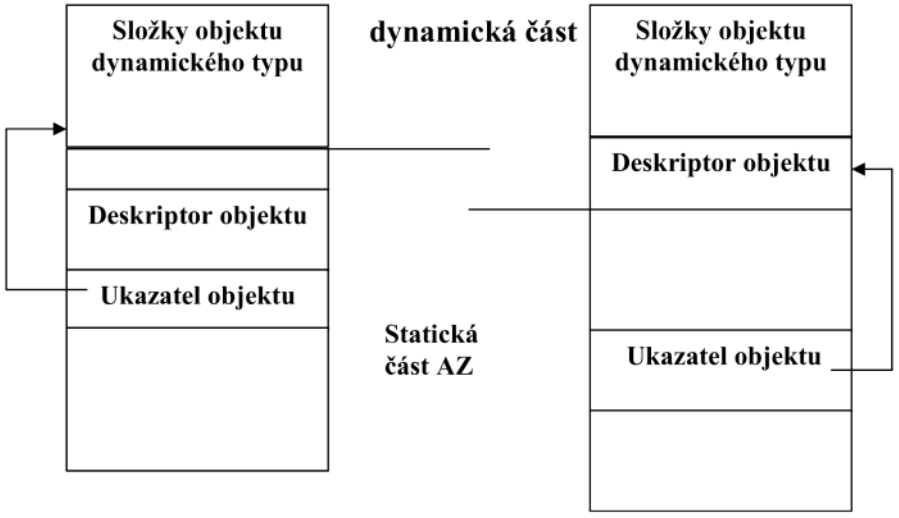
Dynamická adresa proměnné je dvojice (n, p) = displej[n] + p

Page 9

Monday, June 3, 2013 1:08 PM

Objekty s dynamickými typy (typicky pole s proměnnými mezemi)

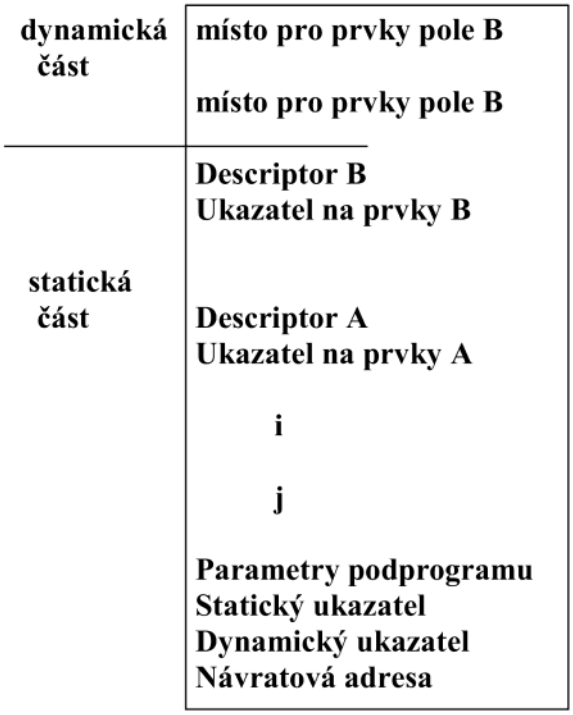
Možnosti struktury aktivačního záznamu s objektem dynamického typu



Deskriptor se vytvoří při překladu, uchovat se ale musí i při výpočtu

Př.3 Aktivačního záznamu s objekty dynamického typu

```
podprogram PRIKLAD;
        int i, j ;
        int A(m .. n);
        int B(p .. q, r .. s,  );
```



Page 10

Monday, June 3, 2013 1:08 PM

Předávání parametrů podprogramům

-hodnotou (C, C++, Java,C#) formální parametr je lokální proměnnou do níž se předá hodnota

-odkazem (C, C++ je-li parametrem pointer, objektové parametry Javy, C# označené ref) předá informaci o umístění skutečného parametru

-výsledkem - formální parametr je lokální proměnnou z níž se předá hodnota do skutečného parametru před návratem z podprogramu slouží jako výstupní parametr

-hodnotou výsledkem (novější Fortran) - kombinace

-jménem – má efekt textové substituce (jako historická zajímavost)

-v případě strukturovaných parametrů

-jsou-li to statické typy ⇒ předá se adresa prvého prvku

-jsou=li to dynamické typy ⇒ předá se ukazatel na descriptor

-je-li parametrem podprogram

-u jazyků nedovolujících hnízdění podprogramů ⇒předá se adresa začátku = pointer

- u jazyků dovolujících hnízdění podprogramů ⇒

spolu s adresou musí předdat i platné prostředí. Jsou různé možnosti co považovat za platné prostředí:

-**mělká vazba** ⇒ platné je prostředí v němž se nachází volání formálního podprogramu

-**hluboká vazba** ⇒platné je prostředí kde je předávaný podprogram definován

-**ad hoc vazba** ⇒platné je prostředí kde je vydán příkaz volání podprogramu jež má za parametr podprogram

Page 11

Monday, June 3, 2013 1:08 PM

Př.4

```
Podprogram  P1( ) {
        Prom x ;
        Podprogram P2 ( ) {
                Vytiskni (x) ; /*co se tady tiskne?*/
        };
        Podprogram P3 ( ) {
                Prom x ;
                x ← 3;
                P4(P2) ;
        };
        Podprgram P4( podprogram Px ) {
                Prom x ;
                x ← 4;
                call Px( );
        }
        x←1;
        P3( ) ;
}
```

prostředí, kde je předávaný podprogram definován

prostředí, kde je vydán příkaz volání s parametrem podprog.

prostředí, v němž je volán formální podprogram

Při mělké vazbě se tiskne … **?**

Při hluboké vazbě se tiskne … **?**

Při ad hoc vazbě se tiskne … **?**

Př.5

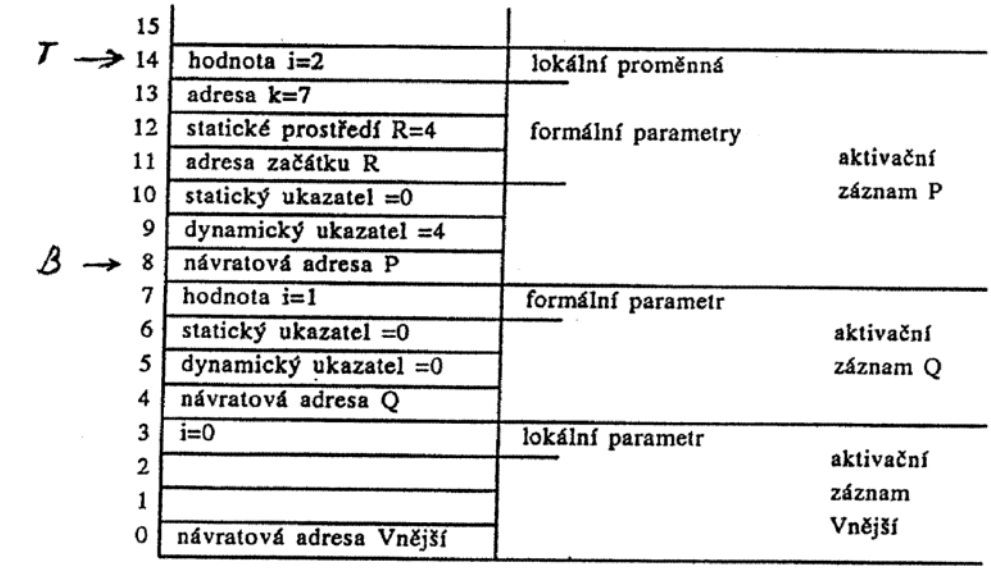
Předpokládejme hlubokou vazbu. Co se vytiskne po spuštění procedury Vnější?

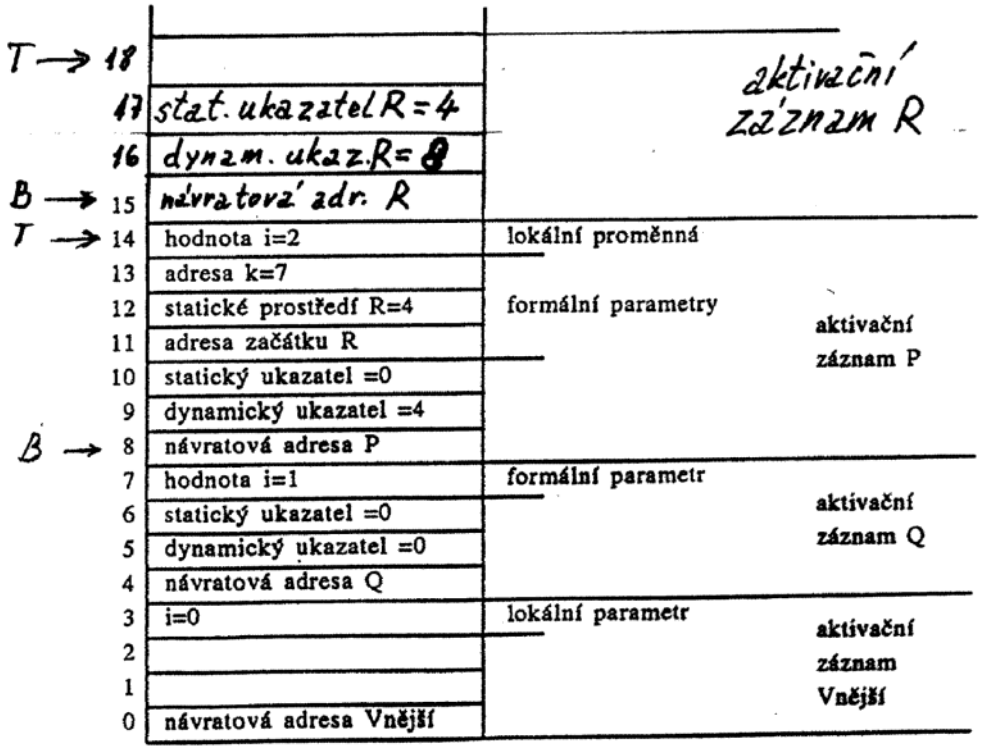
```
podprogram Vnejsi;  {
        prom i:int;
        podprogram P( podprogram FP; prom k:int;)  {
                prom i:int;
                i←k+1;  FP( ); tisk(i);
        }
        podprogram Q(i:int);
                podprogram R ( )  {
                        Tisk(i);
                }
                P(R,i);
        }
        i← 0; Q(i+1);
}
```

Page 12

Monday, June 3, 2013 1:08 PM

Stav před vyvoláním a po vyvolání formálního poprogramu FP z př.5





Page 13

Monday, June 3, 2013 1:08 PM

Přidělování paměti pro paralelní výpočty

Pro uložení AZ paralelního výpočtu nutno použít haldu nebo zobecněný zásobník

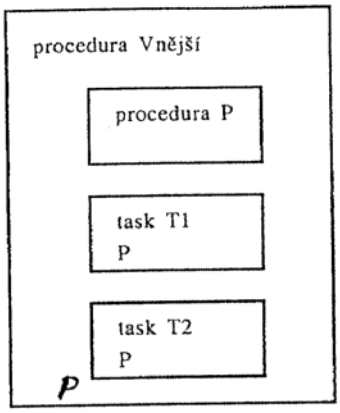
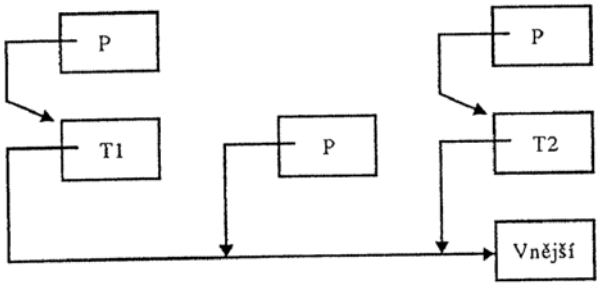
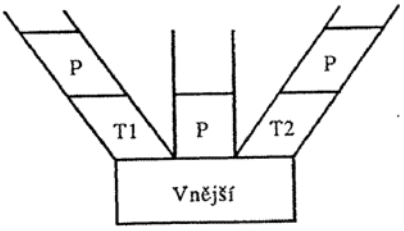


Schéma vnoření bloků programu



Struktura aktivačních záznamů při paralelním výpočtu



Uložení aktivačních záznamů ve zobecněném zásobníku

Page 14

Monday, June 3, 2013 1:08 PM

Př v javovském prostředí