Formální jazyky a překladače

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

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

Zápočet:
Udělen na základě referátu a předvedení zadané modifikace překladače PL0

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

Literatura: web stránky předmětu FJP
Melichar, Češka, Ježek, Rychta: Konstrukce překladačů (ČVUT)
Melichar: Jazyky a překlady (ČVUT)
Molnár a kol.: Gramatiky a jazyky (Alfa)

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

Formální jazyky a překladače - organizace

FJP je šestikreditový předmět doporučený pro 4. ročník *Informatika a výpočetní technika*. K získání zkoušky je zapotřebí splnit požadavky cvičení a napsat zkouškový test. Celkové hodnocení se snadno zjistí z bodového zisku a následující převodní tabulky:

Více než 83 b.	**výborně**
65 – 82 b.	**velmi dobře**
50 – 64 b.	**dobře**
0 – 50 b.	**nevyhověl**

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

Zkouška probíhá písemně. Maximální hodnocení je 55 bodů. Ze zkouškového testu musí student získat alespoň 30 bodů. V polovině semestru mají studenti možnost napsat dobrovolně jednoduchý test z doposud probrané látky.

Využití teorie formálních jazyků

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

Znalost základních principů překladače umožní i vytváření lepších programů v konkrétních jazycích, porozumění výhodám a nevýhodám konkrétních programových struktur.

Formálně je překladač zobrazením:

Překladač: zdrojový jazyk ⟶ cílový jazyk

Činnost assembleru: JSI ⟶ strojový kód

absolutní binární kód /přemístitelný binární kód

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

Činnost interpretu: vyšší progr. jazyk ⟹ výsledky
data

Dávkový překladač batchové zpracování

Konverzační překladač interaktivní

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

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

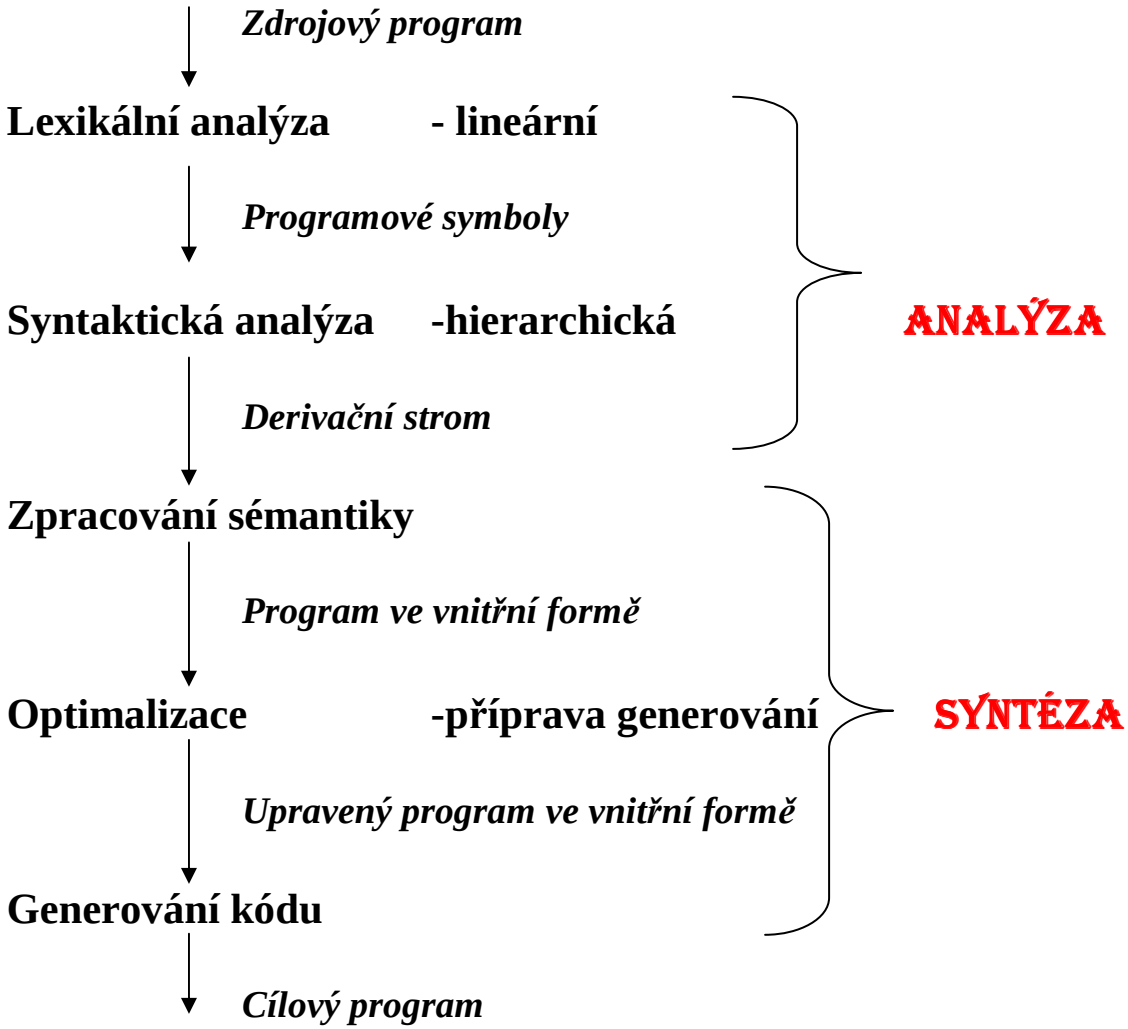
Kaskádní překladač máme již A → B, ale chceme A→C, uděláme B → C. Kdy se to vyplatí? Komplikace - chybová hlášení výpočtu jsou pomíchaná

Optimalizující překladač (možnost ovlivnění optimalizace času / paměti programátorem)

Paralelizující překladač ≈ zjišťuje nezávislost úseků programu

Hlavní části překladače

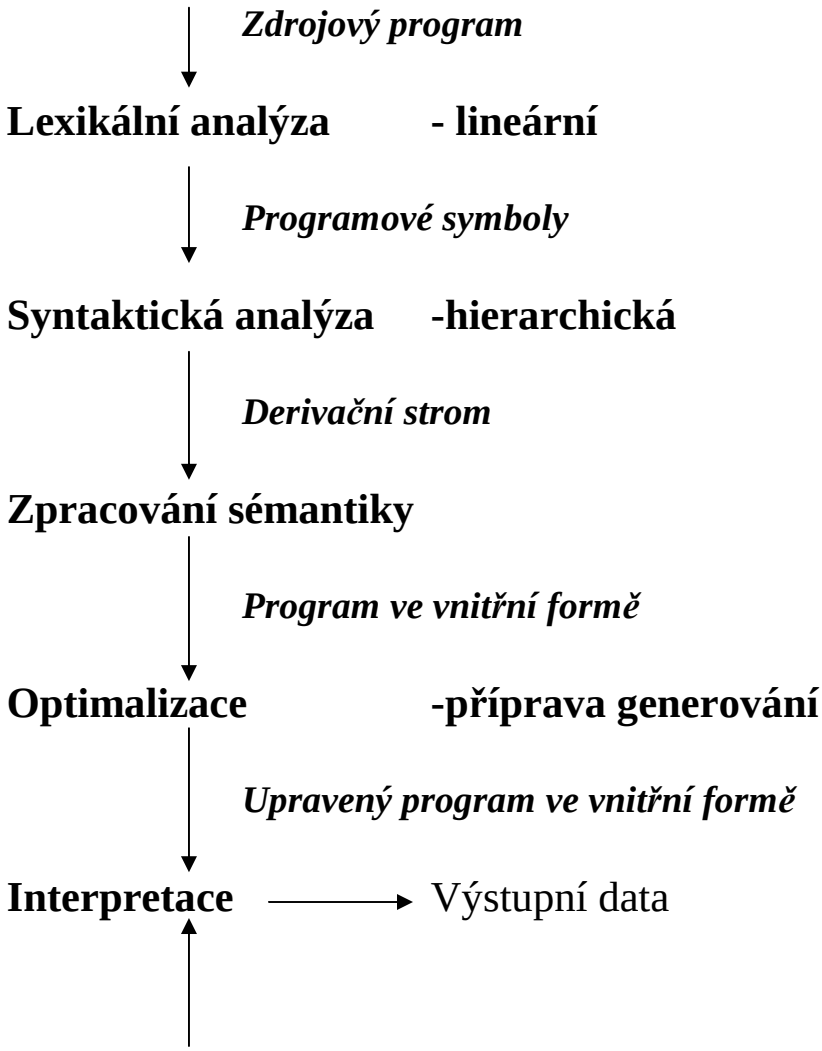
Kompilátor:



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

Výhodou kompilátoru je rychlá exekuce programu

Interpret



Výhodou interpretu je:
eliminace kroků cyklu (Editace→překlad→sestavení→exekuce)

Snazší realizace ladících mechanismů (zachování původních jmen symbolů)

Vícefázový / víceprůchodový překladač

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

Průchod = čtení vstupního řetězce,
zpracování,
zápis výstupního řetězce
(může obsahovat více fází).

Jednoprůchodový překladač =

- všechny fáze probíhají v rámci jediného čtení zdrojového textu programu,
- omezená možnost kontextových kontrol,
- omezená možnost optimalizace,
- lepší možnost zpracování chyb a ladění (pro výuku)

Co má vliv na strukturu překladače

- Vlastnosti zdrojového a cílového jazyka,
- Vlastnosti hostitelského počítače,
- Rychlost/velikost překladače,
- Rychlost/velikost cílového kódu,
- Ladicí schopnosti (detekce chyb, zotavení),
- Velikost projektu, prostředky, termíny.

Testování a údržba překladače

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

Vnitřní jazyky překladače

- **Postfixová notace (operátory bezprostředně následují za svými operandy, pořadí operandů je zachováno)**
  Vyjadřuje precedenci operátorů, nepotřebuje závorky
  Př.1 a + b → a b +
  Př.2 (a + b) * (c + d) → a b + c d + *
  Postfixový zápis nepotřebuje (a nemá) závorky
  Postfix je elegantně vyhodnotitelný zásobníkem:
  1) Čti symbol postfixového řetězce,
  2) Je-li symbolem operand, ulož jej do zásobníku.
  3) Je-li symbolem operátor, proveď jeho operaci nad vrcholem zásobníku a výsledek vlož do zásobníku
  4) Jdi na 1).
  Po přečtení a vyhodnocení celého řetězce je výsledek uložen v zásobníku (princip interpretace).
  Pro př.2

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

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

- **Prefixová notace (operátory bezprostředně předchází operandy, pořadí operandů je zachováno)**
  Vyjadřuje precedenci operátorů, nepotřebuje závorky
  a + b → + a b
  (a + b) * (c + d) → * + a b + c d
  Prefixový zápis nemá závorky
  !Pozor, není to vždy zrcadlový obraz operátorů z postfixu

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

Zkusme na tabuli př. A = - B * C + D

- **Víceadresové instrukce (čtveřice / trojice)**
  Čtveřice **operátor, operand, operand, výsledek**
  Např. +, a, b, Vysledek
  Potřeba přidělovat paměť pomocným prom.

Př 2) **(a + b) * (c + d)**

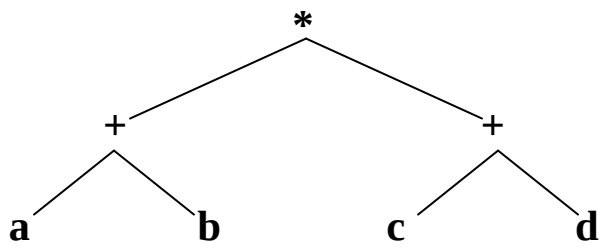
tvar	význam
+, a, b, Pom1	Pom1 = a + b
+, c. d. Pom2	Pom2 = c + d
*, Pom1, Pom2, Vysl	Vysl = Pom1+ Pom2

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

Př 2)

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

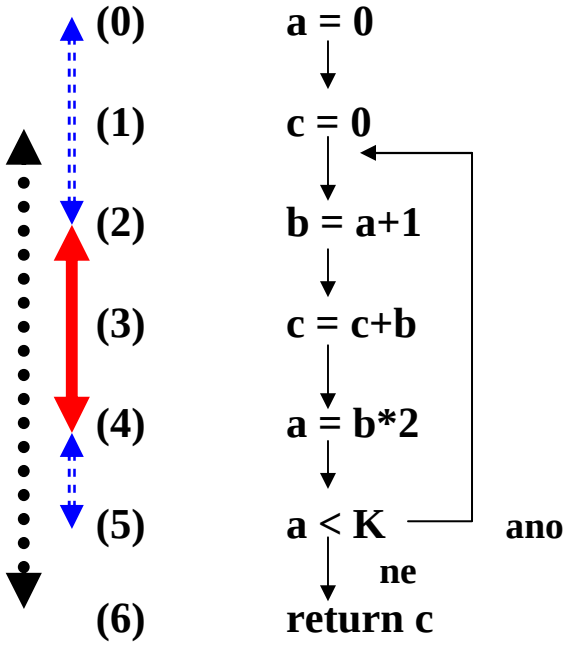
Vyjadřují abstraktní syntaktický strom



Optimalizace

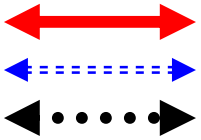
- **Optimalizace cyklů**
- **Redukce počtu registrů**
- **…a další**

Př optimalizace paměti.



program

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



živé b (2) → (3) → (4)

živé a (0) → (1) → (2), (4) → (5) → (2)

živé c (1) → (2) → (3) → (4) → (5) → (2)

(6)

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

-zjištění živých a neživých proměnných v data flow diagramu,

-vytvoření interferenčního grafu,

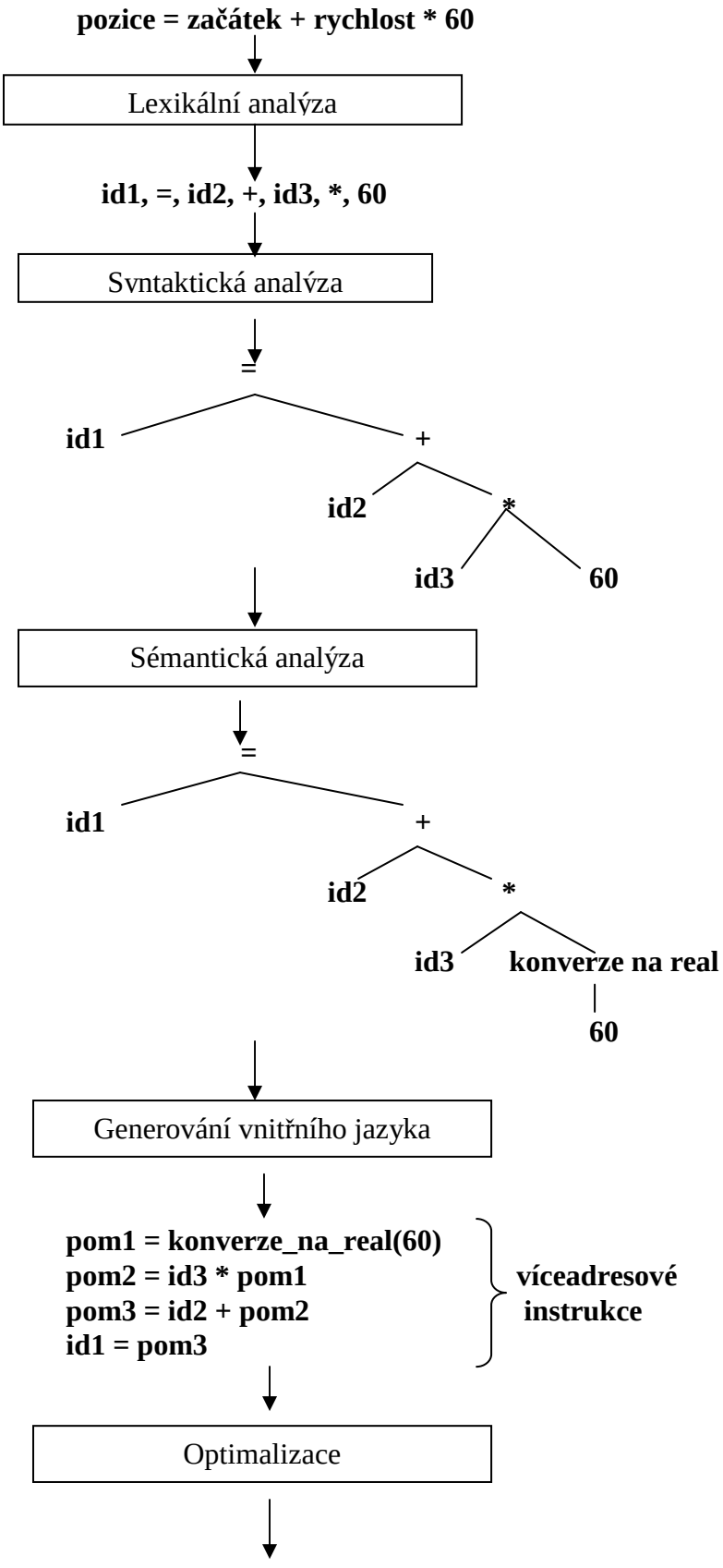
-barvení grafu.

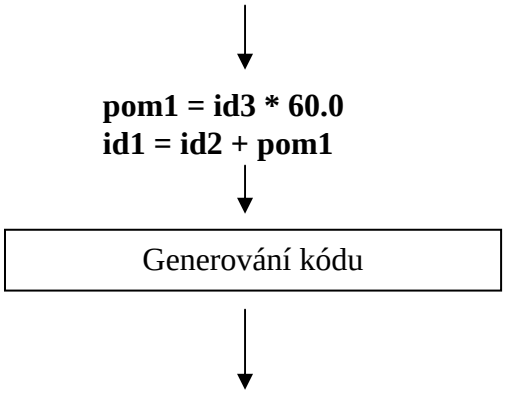
počet potřebných barev = počet potřebných registrů

Matice interferencí

	a	b	c
a			x
b			x
c	x	x	

Př. překladu příkazu

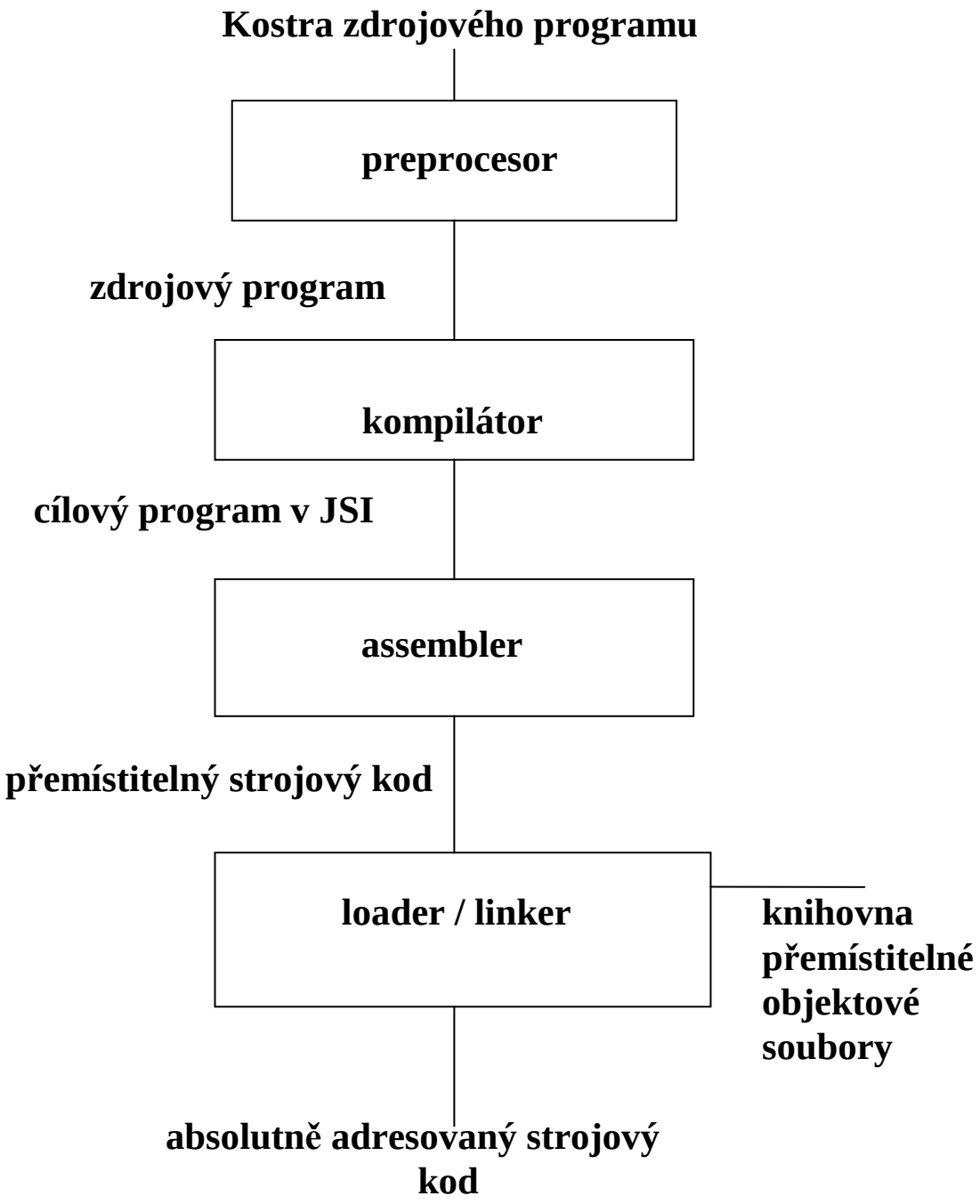




MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1

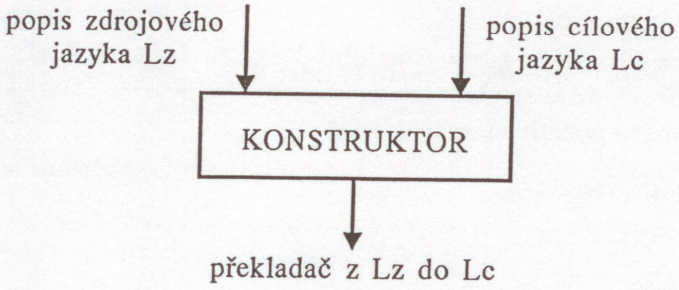
Tabulka symbolů

1	**pozice**	**…**
2	**zacatek**	**…**
3	**rychlost**	**…**
4		
5		

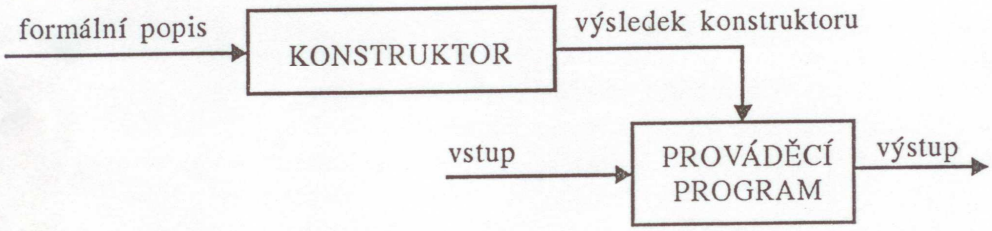


Obr. Systém zpracování jazyka

Principiální možnost automatizace konstrukce překladače

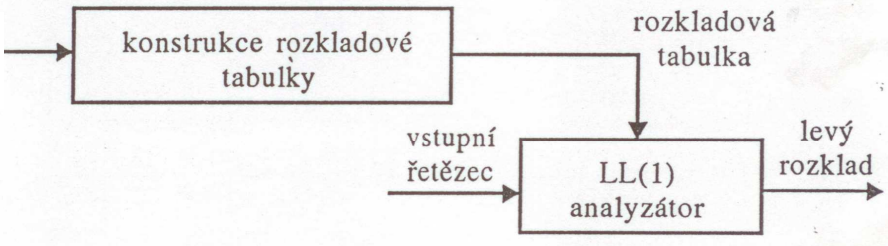


Obr. 1.12: Struktura dokonalého systému pro konstrukci překladače



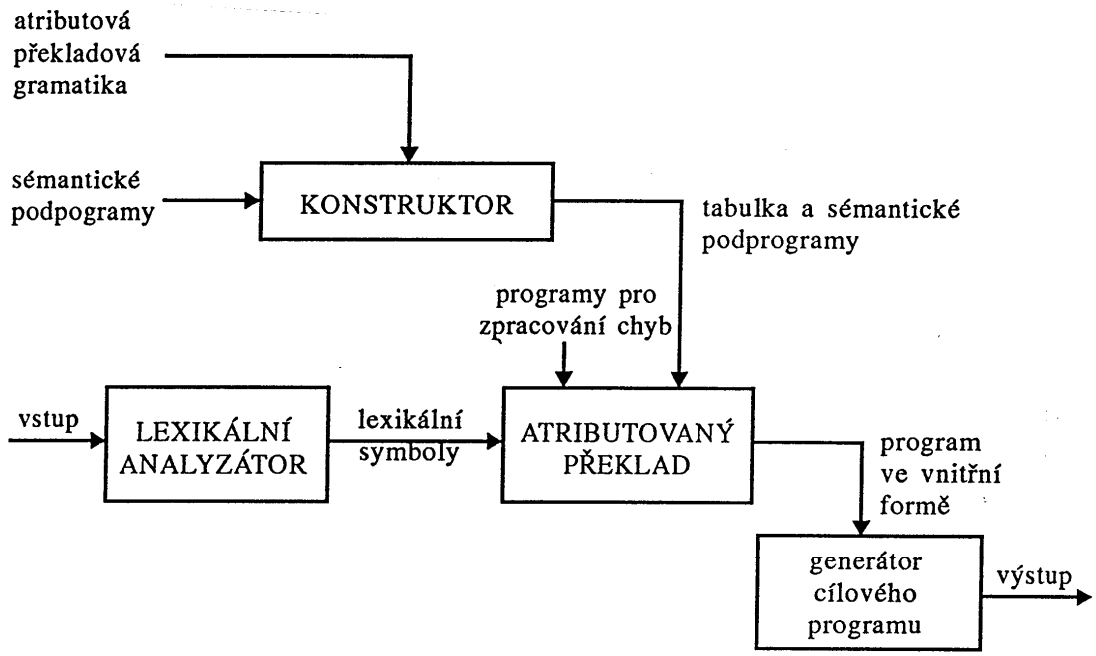
Obr. 1.13: Dvojice *konstruktor—prováděcí program*

Bezkontextová gramatika

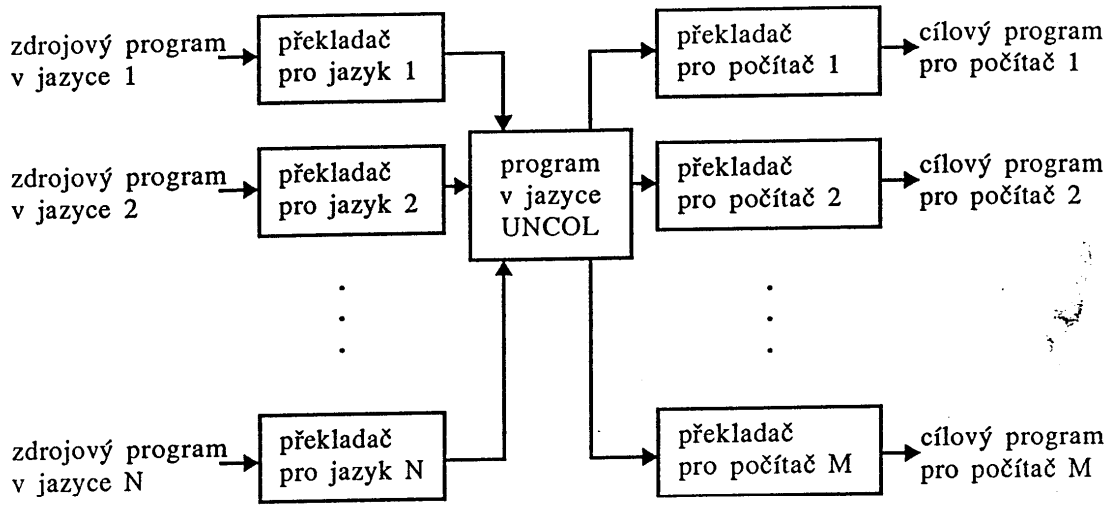


Obr. 1.14: Systém pro konstrukci $LL(1)$ analyzátoru

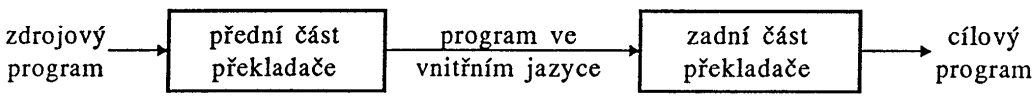
Jaké prostředky k tomu máme a co bychom chtěli



Obr. Systém automatické konstrukce syntaktického analyzátoru s připojeným zpracování sémantiky



Obr. Základní myšlenka univerzálního vnitřního jazyka UNCOL



Obr. Přední a zadní část překladače

http://osteele.com/tools/reanimator/ animace regulárních výrazů
http://osteele.com/tools/rework/# Javascript, Python, PHP, Ruby
http://dinosaur.compilertools.net/ Lex, Flex, Yacc, Bizon stránky

The Lex & Yacc Page

Overview | Lex | Yacc | Flex | Bison | Tools | Books

OVERVIEW

A compiler or interptreter for a programminning language is often decomposed into two parts:

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

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

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

1. Split the source file into tokens (*Lex*).
2. Find the hierarchical structure of the program (*Yacc*).

- A First Example: A Simple Interpreter

LEX

Lex - A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt

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

- Online Manual
- PostScript
- Lex Manual Page

YACC

Yacc: Yet Another Compiler-Compiler

Stephen C. Johnson

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an ``input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

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

- Online Manual
- PostScript
- Yacc Manual Page

FLEX

Flex, A fast scanner generator

Vern Paxson

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

- Online Manual
- PostScript
- Flex Manual Page
- Download Flex from ftp://prep.ai.mit.edu/pub/gnu/

BISON

Bison, The YACC-compatible Parser Generator

Charles Donnelly and Richard Stallman

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

- Online Manual
- PostScript

- Bison Manual Page
- Download Bison from ftp://prep.ai.mit.edu/pub/gnu/

TOOLS

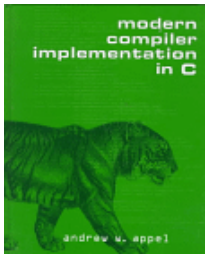
Other tools for compiler writers:

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

BOOKS

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

Compilers: Principles, Techniques, and Tools
Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman
Addison-Wesley Pub Co
ISBN: 0201100886



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

Lexikální analýza
(Obsah)

1. **Rekapitulace potřebných znalostí**
   -Regulární jazyky, regulární výrazy
   -Pravé lineární gramatiky
   -Konečné automaty (tabulka přechodů, stavový diagram, stavový strom)
   -Převod gramatika – konečný automat
   -Nedeterministický konečný automat, převod na deterministický

2. **Levé lineární gramatiky**

3. **Korespondence gramatik typu3 a konečných automatů**

4. **Vytváření derivačního stromu v případě lineárních gramatik**

5. **Regulární atributované překladové gramatiky**

6. **Princip lexikální analýzy**

7. **Konstruktory lexikálního analyzátoru LEX, FLEX**

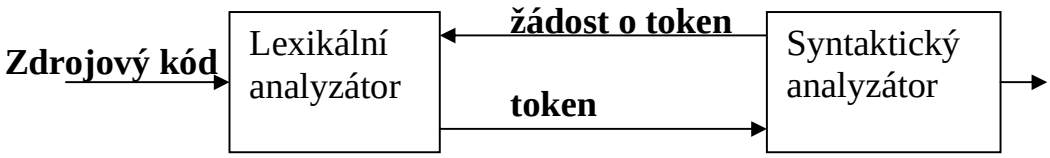
Úkoly lexikálního analyzátoru

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

Proč je LA samostatnou částí

- **Jednodušší návrh překladače**
- **Zlepšení efektivity překladu**
- **Lepší přenositelnost**

Lexikální analyzátor rozpoznává a zakóduje lexikální symboly jazyka (lexémy anglicky tokens)



Lexikální symboly jsou regulárním jazykem

- **Regulární jazyk**

  Lze definovat gramatikou typu 3

  G = (N, T, P, S) kde P mají tvar

  X → w Y nebo X → w kde w ϵ T^*

  (velkými písmeny označujeme neterminální symboly)

  Př1. **S → 1A**
  A → 0A | 1

  Př2. **S → 1A | 1B**
  A → 0A | 0
  B → 1B | 1

  nebo konečným automatem

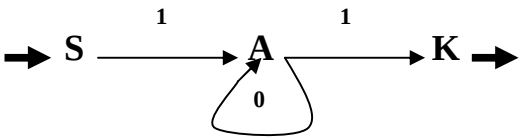
  formální popis je pětice KA = (Q, X, δ, q_0, F)

  způsoby reprezentace přechodové funkce

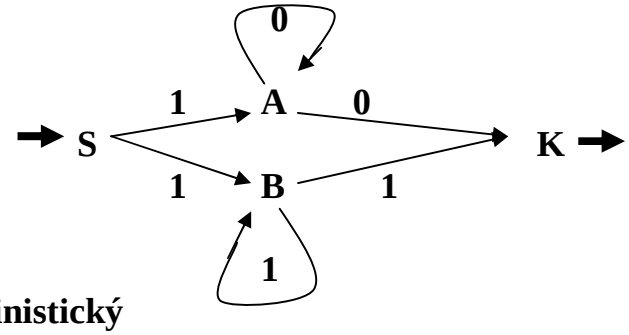
  -tabulka přechodů,

  -stavový diagram,

Př1



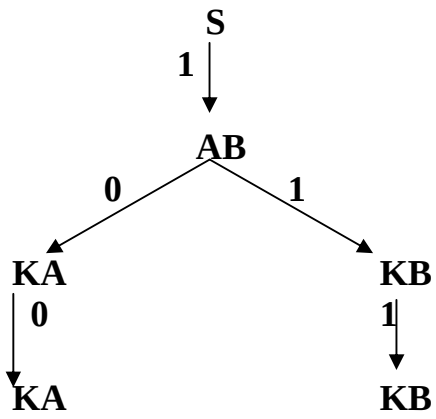
Př2



Je nedeterministický

-stavový strom

Př2



Teď je již deterministický

nebo regulárním výrazem

Př2 $1 (0^{*} 0 + 1^{*} 1)$

dtto je $1 (0^{+} + 1^{+})$

? ale co $1 (0^{n} + 1^{n})$ **pro n>=0 ?**

- **Pumping lemma: Nechť L je regulární množina (regulární jazyk), pak existuje konstanta p taková, že je-li w ϵ L a |w| >= p, pak w lze zapsat ve tvaru xyz, kde 0 < |y| <= p a $xy^{i}z$ ϵ L pro všechna i >= 0**

? je regulárním jazykem $1\ 0^{n}\ 1$ **pro n >= 0**

? je regulárním jazykem $1^{n}\ 0^{n}$ **„**

? je regulárním jazykem $1^{n}\ 2^{n}\ 3^{n}$ **„**

? je regulárním jazykem $(1\ 2\ 3)^{n}$ **„**

Konfigurace automatu je dvojice (stav, ještě nezpracovaný vstup)

Počáteční konfigurace (S, věta), kde S je počáteční stav

Koncová konfigurace (K, e), kde K je koncový stav a e je prázdný řetězec

├ je znak přechodu mezi konfiguracemi

Př. Analýza věty 1 0 0 1 jazyka $1\ 0^{n}\ 1$ **(př.1)**

(S, 1001) ├ (A, 001) ├ (A, 01) ├ (A, 1) ├ (K, e)

Levé a pravé lineární gramatiky

Pravá lineární:
G = (N, T, P, S) kde P mají tvar $X \rightarrow w\,Y$ $w \in T^*$
$X \rightarrow w$

Levá lineární:
G = (N, T, P, S) kde P mají tvar $X \rightarrow Y\,w$ $w \in T^*$
$X \rightarrow w$

Lze převést na tvar $X \rightarrow Y\,a$ **a je term.symbol**
$X \rightarrow a$ **příp.** $X \rightarrow e$

Každou lineární gramatiku lze převést na regulární tvar

Konstrukce ekvivalentního KA pro levou regulární gramatiku:

- **Neterminálnímu symbolu odpovídá stav**
- **Počáteční stav nepatří do N (je jím i stav A, pro nějž $A \rightarrow e \in P$)**
- **Každému pravidlu odpovídá větev takto:**

1) **Z Y do X označená a, je-li $X \rightarrow Ya \in P$**
2) **Z počátečního stavu do X označená a. je-li $X \rightarrow a \in P$**
3) **Koncovým stavem je počáteční symbol gramatiky**

Př. Konstruujte KA pro G = (N, T, P, Z) , kde P: $Z \rightarrow U0 \mid V1$
$U \rightarrow Z1 \mid 1$
$V \rightarrow Z0 \mid 0$

S U

Jak by vypadala ekvival. pravá lin.g.?
(má poč. symbol S)

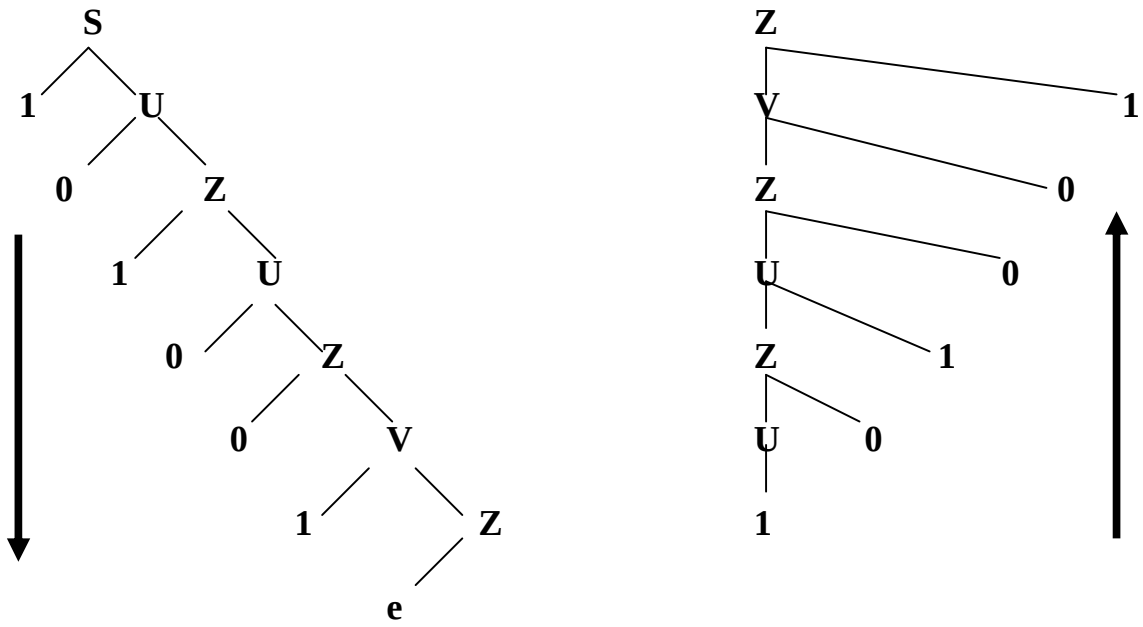
Doplňme

V Z

Generování věty 101001 (symbol => znamená derivuje)

↓ **S => 1U => 10Z => 101U => 1010Z => 10100V => 101001Z => 101001**
↑ **Z => V1 => Z01 => U001 => Z1001 => U01001 => 101001**

Graficky zachycuje derivaci derivační strom



Vstupující řetězec vždy čteme zleva doprava (všimněte si jak se liší konstrukce derivačního stromu, princip expanze neterminálu při ↓versus redukce na neterminál při ↑, vstup se vždy zpracovává/čte z leva)

Př.Zapište gramatiku identifikátoru a) pravou, b)levou lineární gramatikou

Konstruujte ekvivalentní automat

Zkuste derivovat nějakou větu a vykreslit její derivační strom

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

$X \rightarrow a\ w'\ Y$, 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

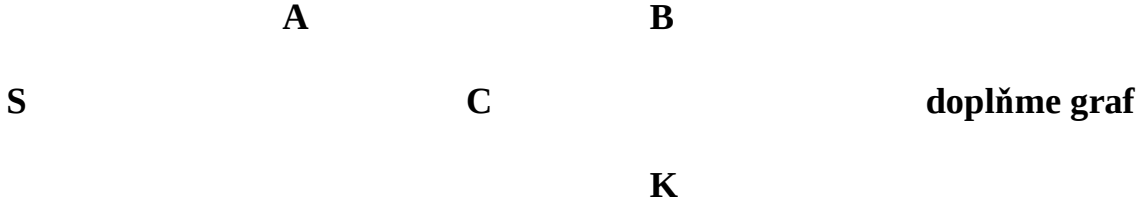
=> i i‘ * i i‘ *‘ + i i‘ +‘

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



doplňme graf

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 → č Z	Z.hodnota = ord(č.kód) – ord('0')
Z^0 → 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

<u>Princip lexikálního analyzátoru</u> (Nalezení a rozpoznání lexikálního symbolu)

Třídy symbolů:

- **Identifikátory**
- **Klíčová slova (rezervované identifikátory)**
- **Celá čísla**
- **Jednoznakové omezovače**
- **Dvouznakové omezovače**

Gramatický popis tříd symbolů:

<identifikátor> → písmeno <id>
<id> → písmeno <id> | číslice <id> | e
<klíčové slovo> → begin | end | do | while
<celé číslo> → číslice | číslice <celé číslo>
<jednoznakový omezovač> → + | - | / | * | (|)
<dvouznakový omezovač> → // | ** | :=

poznámky tvaru: /* poznámka */

? co je počátečním symbolem?

<Symbol> → <identifikátor> |
<celé číslo> |
<jednoznakový omezovač> |
<dvouznakový omezovač>

…

zakódování symbolů zvolme např.:

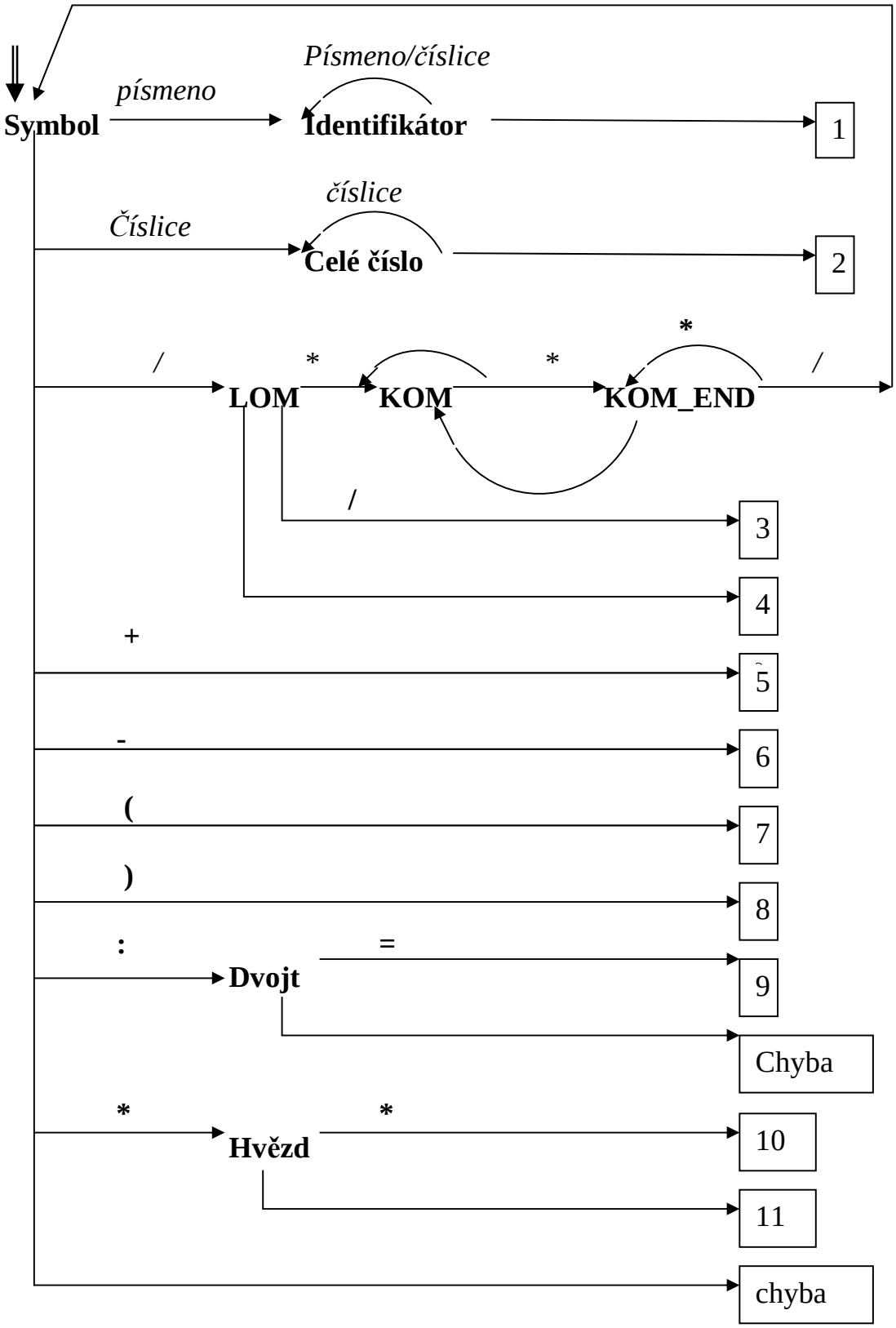
symbol	kód	symbol	kód	symbol	kód
identifikátor	1	celé číslo	2	//	3
/	4	+	5	-	6
(	7	)	8	:=	9
**	10	*	11	begin	12
end	13	do	14	while	15

? jak vypadá konečný automat?

- **Zpracování začíná prvým dosud nezpracovaným znakem ze vstupu,**
- **Zpracování končí, je-li automat v koncovém stavu a pro další vstupní znak již neexistuje žádný přechod**
- **Pro každou kategorii předpokládáme samostatný koncový stav,**
- **Neohodnocená větev se vybere, pokud vstupujícímu znaku neodpovídá žádná z ohodnocených větví**

Při zpracování sémantiky symbolů

- **Hodnoty atributů se vypočtou z lexému**
- **Klíčová slova / rezervované identifikátory rozlišíme za pomoci tabulky klíčových slov.**


Písmeno/číslice
písmeno
Symbol
Identifikátor
1
číslice
Číslice
Celé číslo
2
*
/
*
*
/
LOM
KOM
KOM_END
/
3
4
+
5
-
6
(
7
)
8
:
=
Dvojt
9
Chyba
*
*
Hvězd
10
11
chyba

Sémantické zpracování lexikálních elementů

Kód lexémů představuje informaci o druhu lexému, ne plně o jeho významu.

Rozdíl +, /,
do, while
1415, x1, alfa

LA musí předat i atributy lexému, tj.
- u čísel jejich hodnotu
- u identifikátorů textový tvar (či adresu / ukazatel do tabulky)

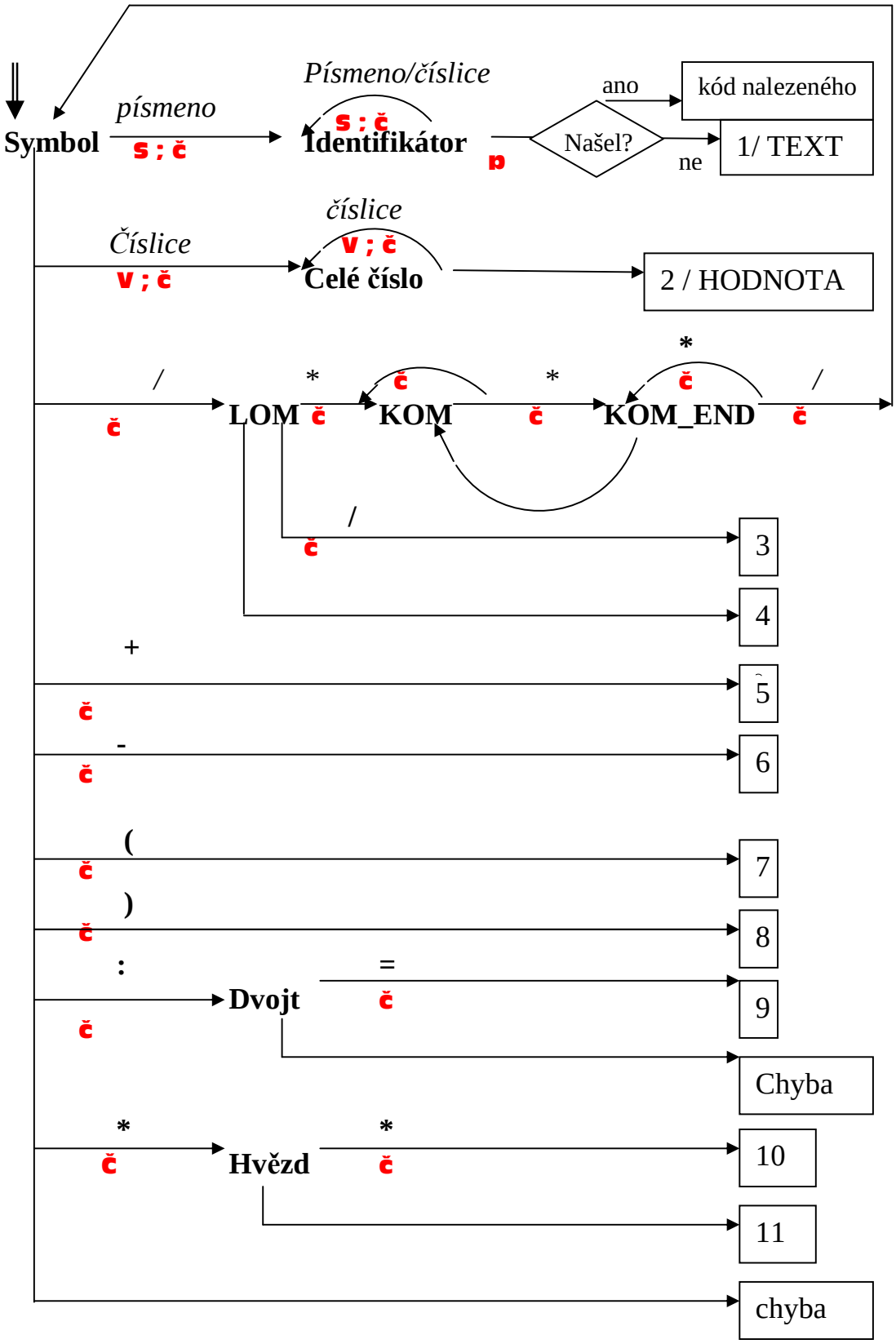
Předp. Identifikátor předáván jako dvojice 1, text
Číslo „ „ „ „ 2. hodnota

Např. do /*dokud te to bavi*/ alfa := 10 * (x + y)
převede na
14, 1, alfa, 9, 2, 10, 11, 7, 1, x, 5, 1, y, 8
nebo
14, -, 1, alfa, 9, -, 2, 10, 11, -, 7, -, 1, x, 5, -, 1, y, 8, -

Automat LA bude rozšířen o funkce

- **č** ČTI čte jeden znak zdrojového textu (posouvá hlavičku KA)
- **s** SLOŽ zřetězuje znaky do proměnné TEXT
- **p** PROHLEDEJ hledá v tabulce rezervovaných slov a v případě nalezení vrací jeho kód
- **v** VYPOČTI po znacích vyčísluje hodnotu konstanty do proměnné HODNOTA

? jak je zařadit do diagramu ?


Symbol
písmeno
s ; č
Písmeno/číslice
s ; č
Identifikátor
p
Našel?
ano
kód nalezeného
ne
1/ TEXT
Číslice
v ; č
číslice
v ; č
Celé číslo
2 / HODNOTA
/
č
*
č
LOM
č
KOM
č
*
č
KOM_END
*
č
/
č
3
4
+
č
5
-
č
6
(
č
7
)
č
8
:
č
Dvojt
=
č
9
Chyba
*
č
Hvězd
*
č
10
11
chyba

Nejednoznačnosti v lexikální analyze

Nastává v případě, kdy jeden symbol je prefixem jiného symbolu (== apod.)

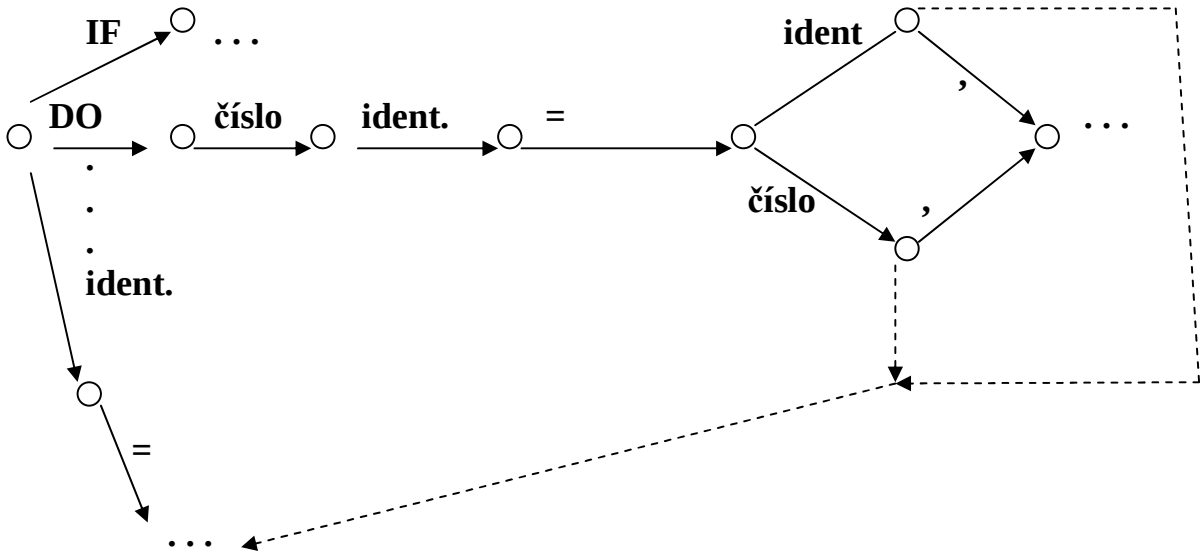
Pravidlo “hledej nejdelší symbol”

Nedokonalosti některých jazyků

Fortranské číslo versus relace 123 . EQ. Y
Nutno ukládat znaky do pomocného pole

Příkaz cyklu DO 10 I = 1 , 5

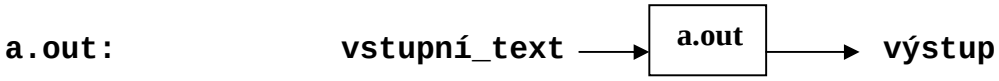
Vyžaduje nápovědu od syntaktického analyzátoru



LEX/FLEX (Unix/Linux)

Pro Windows lze stahnout "complete package, except sources" ze stránky:
http://gnuwin32.sourceforge.net/packages/bison.htm
http://gnuwin32.sourceforge.net/packages/flex.htm

Generuje program v jazyce C do souboru lex.yy.c, který definuje funkci yylex(). Po přeložení generuje proveditelný kód. Manuál máte v souboru Flex.pdf




```
Obecný tvar vstupního souboru pro Lex:
      {definice použité v regulárních výrazech a C deklarace}
      %%
      {pravidla v podobě regulárních výrazů a příslušných akcí}
      %%
      {doplňkové procedury}

-Definice – zahrnují deklarace  proměnných,
                                konstant,
                                regulárních definic.

-Pravidla mají tvar –     p1    {akce1 v C notaci}
                          P2    {akce2   "       }
                                …
                          pn    {akceN   „       }

            pi jsou regulární výrazy
            {akcei} jsou programové fragmenty

-Doplňkové procedury jsou pomocné, mohou obsahovat C rutiny
volané akcemi
```

Regulární výrazy v pravidlech mohou mít podobu:

je-li	**c**	**jeden znak**
	r	**reg. výraz**
	s	**řetězec**
	i	**identifikátor**

pak výrazu	**odpovídá**	**např.**
c	**libov. neoperátorový znak c**	**a**
\c	**znak c literálně**	*****
"s"	**řetězec s literálně**	**"**"**
.	**libov. znak mimo nový řádek**	**a.*b**
^	**začátek řádky**	**^abc**
$	**konec řádky**	**abc$**
[s]	**libov. znak z s**	**[abc]**
[x-z]	**znaky x, y, . . . z.**	**[0-9]**
[^s]	**" " " není-li z s**	**[^abc]**
r*	**nula nebo více r**	**a***
r+	**jeden nebo více r**	**a+**
r?	**nula nebo jeden r**	**a?**
r{m,n}	**m až n výskytů r**	**a{1,5}**
r1r2	**r1 pak r2**	**ab**
r1|r2	**r1 nebo r2**	**a|b**
(r)	**r**	**(a|b)**
r1/r2	**r1 je-li n sledováno r2**	**abc/123**
{i}	**překlad i z definiční sekce**	**{PISMENO}**

yylval	**proměnná pro předání tokenu do Yacc (ten provádí synt. analýzu)**
yytext	**proměnná obsahující text odpovídajícího reg.výrazu**
yyleng	**" " počet znaků "**
yyless(n)	**ubere n znaků z yytext[]**
yymore()	**přidá k obsahu yytext[] další koresp. část textu**
REJECT	**přejde na další pravidlo bez změny obsahu yytext[]**

Příklad

```
%{
        /* definice manifestovych konstant
        LT, LE, EQ, NE, GT, GE, IF, THEN, ELSE,
        ID, NUMBER, RELOP */
%}

/* regularni definice */
delim     [ \t\n]
ws        {delim}*
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

%%
{ws}            {/* zadna akce ani navrat */}
if              {return(IF);}
then            {return(THEN);}
else            {return(ELSE);}
{id}            {yylval=install_id(); return(ID);}
{number}        {yylval=install_num(); return(NUMBER);}
"<="            {yylval=LE; return(RELOP);}
"="             {yylval=EQ; return(RELOP);}
"<>"            {yylval=NE; return(RELOP);}
">="            {yylval=GE; return(RELOP):}
“<”             {yylval=LT; return(RELOP);}
“>”             {yylval=GT; return(RELOP);}
%%

install_id() {
        /* vlozi do tabulky symbolu lex.elem.,jehoz prvy
        znak je urceny v yytext a delka je v yyleng.
        Vracenou hodnotou je ukazatel do tab.sym. NEROZEPSANA
        */
}
install_num() {
        /*podobne, pro instalaci cisla*/
}
```

Prostředky pro regulární výrazy jiných programovacích jazyků jsou z Lex

viz http://osteele.com/tools/rework/#

Př.1) [a-zA-Z][0-9a-zA-Z]*

JavaScript
```
"nejaky text".match(/[a-zA-Z][0-9a-zA-Z]*/g)
re.exec("nejaky text")
```
PHP
```
preg_match_all('/[a-zA-Z][0-9a-zA-Z]*/', "nejaky text", $match)
```
Python
```
re.findall(r'[a-zA-Z][0-9a-zA-Z]*', "nejaky text")
```
Ruby
```
"nejaky text".scan(/[a-zA-Z][0-9a-zA-Z]*/)
```

Použito v Python
>>> re.findall(r'[a-zA-Z][0-9a-zA-Z]*', 'ab1 nic 44')
['ab1', 'nic']

Př. 2) ([+-]?\d*\.\d+([eE][+-]?\d+)?)

JavaScript
```
"-2.33e-2alfa11beta12e3?.12E3".replace(/([+-]?\d*\.\d+([eE][+-]?\d+)?)/g, "'expcislo' ")
```
PHP
```
preg_replace('/([+-]?\\d*\.\d+([eE][+-]?\d+)?)/', "'expcislo' ", "-2.33e-2alfa11beta12e3?.12E3")
```
Python
```
re.sub(r'([+-]?\\d*\.\d+([eE][+-]?\d+)?)', "'expcislo' ", "-2.33e-2alfa11beta12e3?.12E3")
```
Ruby
```
"-2.33e-2alfa11beta12e3?.12E3".gsub(/([+-]?\\d*\.\d+([eE][+-]?\d+)?)/, "'expcislo' ")
```

Použito v Python
>>> re.sub(r'([+-]?\d*\.\d+([eE][+-]?\d+)?)', "'expcislo' ", "-2.33e-2alfa11beta12e3?.12E3")
"'expcislo' alfa11beta12e3?'expcislo' "
>>> re.findall(r'([+-]?\d*\.\d+([eE][+-]?\d+)?)', "-2.33e-2alfa11beta12e3?.12E3")
[('-2.33e-2', 'e-2'), ('.12E3', 'E3')]

Syntaktická analýza metodou rekurzivního sestupu

Základní vlastnosti

- **Zvládá práci s bezkontextovými gramatikami BKG (nutné pro popis syntaxe programovacích jazyků, nejsou regulární). Obecný tvar pravidel:**

  $A \rightarrow \alpha$ kde $\alpha \in (N \cup T)^*$

- **Deterministická analýza shora dolů (sestup)**
- **Metoda vyjádřená vzájemně se volajícími podprogramy (rekurzivní procedury)**

Pozn.:Obecně je analýza BK jazyka úloha řešitelná metodou s návratem s kubickou složitostí (když se nepodaří generovat/akceptovat daný řetězec, vrátíme se a zkusíme jinou možnost). Pokud ji zvládneme rekurzivním sestupem, pak je složitost lineární.

Princip

- **Každému neterminálnímu symbolu A odpovídá procedura A**
- **Tělo procedury je dáno pravými stranami pravidel pro A**

  A → X11 X12 … X1n | X21 X22 ... X2m | ... | Xp1 Xp2 ... Xpq

  pravé strany musí být rozlišitelné na základě symbolů vstupního řetězce aktuálních v okamžiku uplatnění příslušné pravé strany

- **Je-li rozpoznána pravá strana Xi1 Xi2 ... Xik,**

  pak prováděj pro j = 1 .. k

  1. **Je-li symbol Xij neterminální, vyvolá se v A proceduru Xij**
  2. **Je-li " Xij terminální, ověří A přítomnost Xij ve vstupním řetězci a zajistí přečtení dalšího symbolu ze vstupu**

- **Rozpoznané pravidlo analyzátor oznámí (např. výpisem čísla pravidla)**
- **Chybnou strukturu vstupního řetězce oznámí chybovým hlášením**

Pro rozpoznání správné pravé strany musí platit:

-řetězce derivovatelné z pravých stran začínají různými terminálními symboly

-při prázdné pravé straně se musí navíc lišit i od těch terminálních symbolů, které se mohou vyskytnout v derivacích za neterminálem z levé strany pravidla.

Např. příkaz může začínat *if, while, do, identifikátorem, call,* … tím lze rozlišovat,

ale jak poznat neúplný podm.příkaz od úplného (s *else*)? Prodiskutujte to.

Př. Gramatika přiřazování (použijeme metasymboly opakování { + T } a { * F } tzv. iterační zápis gramatiky)

(1,2) S → V = E | if E then S Z

(3,4) Z → else S | e

(5) E → T { + T }

(6) T → F { * F }

(7,8) F → (E) | V

(9) V → a I

(10,11) I → (E) | e

-Zkuste ji napsat i normálně (bez metasymbolů) a diskutujte možný problém s pravidly

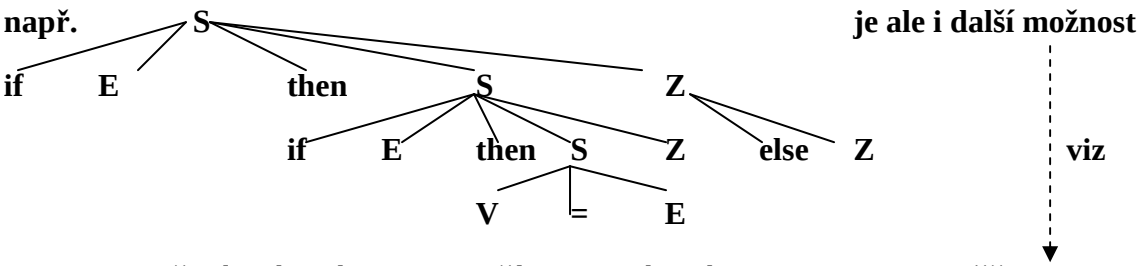
E → T | E + T T → F | T * F

-Zjistěte, jak vypadají množiny *first* terminálních symbolů, kterými začínají řetězce odvoditelné z jednotlivých pravých stran pravidel (dovolí to provést výběr té správné pr.strany na kterou se má expandovat neterminál korespondující dané proceduře)

-Zjistěte, jak vypadají množiny *follow* terminálních symbolů, které následují ve vstupu po provedení příslušné procedury a zkuste si jak vypadá struktura věty

if a then if a then a = a else a = a

Symbol	first	follow
S Z E		
T F V		
I		



Tohle je vnoření neúplného podm. příkazu do úplného, gramatika umožňuje i opak

Lexikální analýzu bude provádět procedura CTI

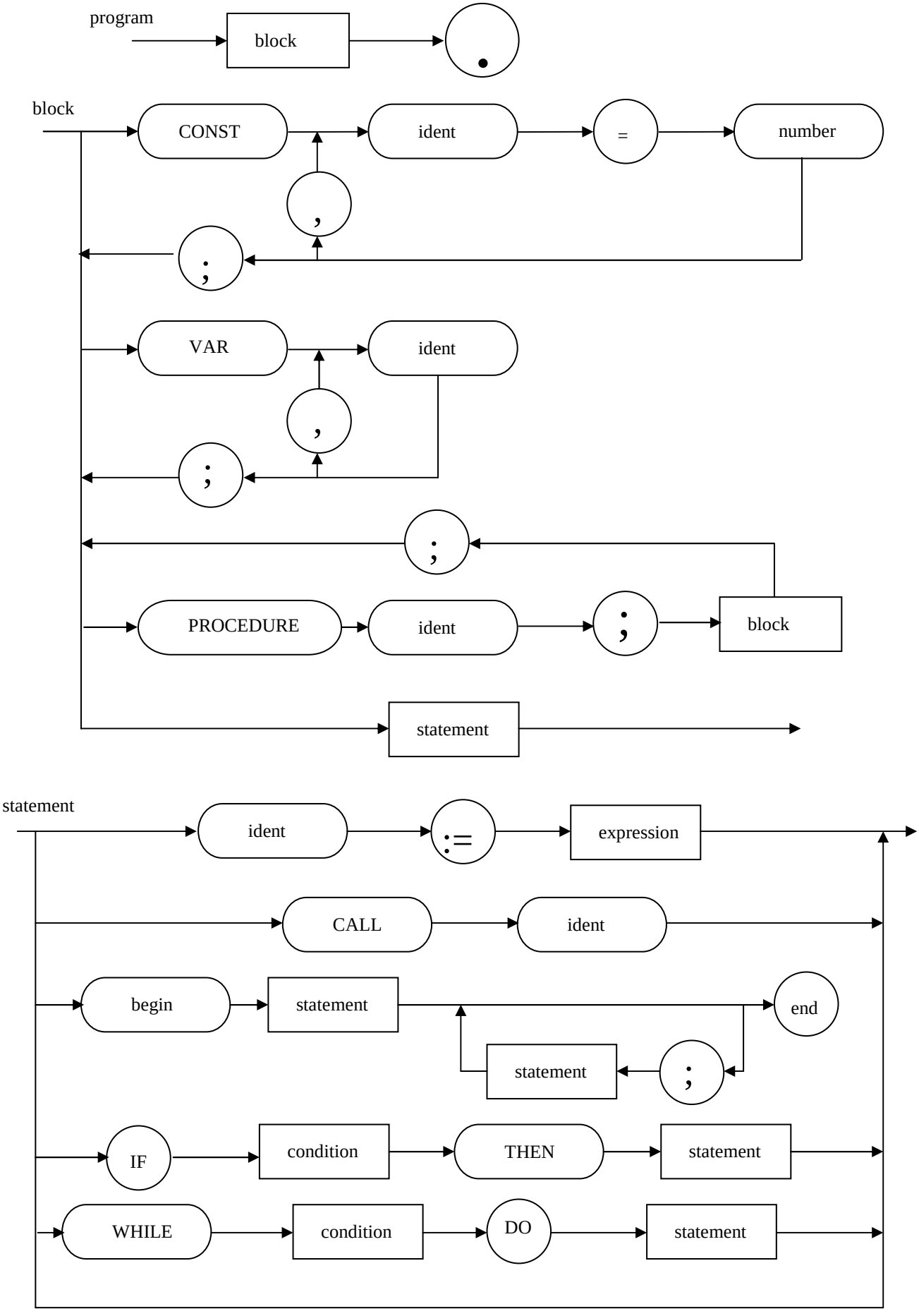
Hlášení chyb bude provádět procedura CHYBA

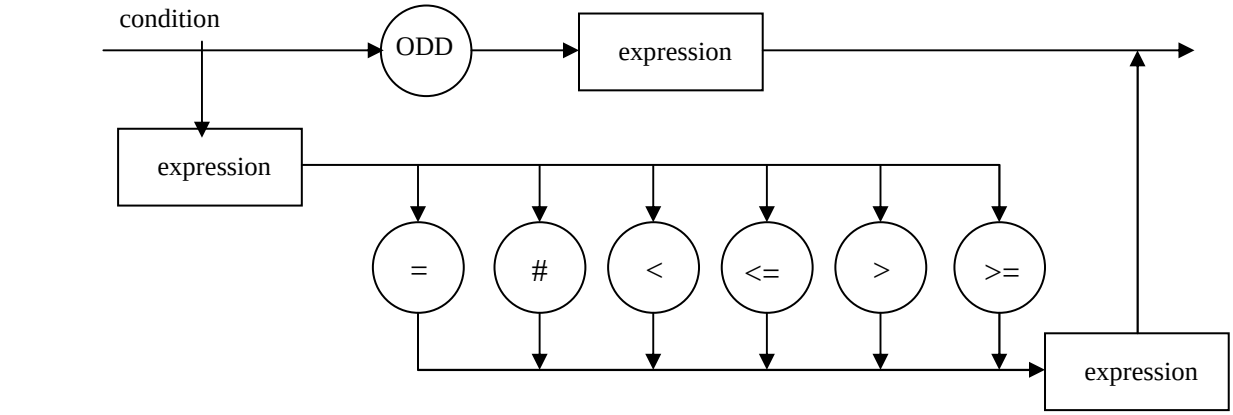
Posloupnost přepisovacích pravidel vypisuje procedura TISK

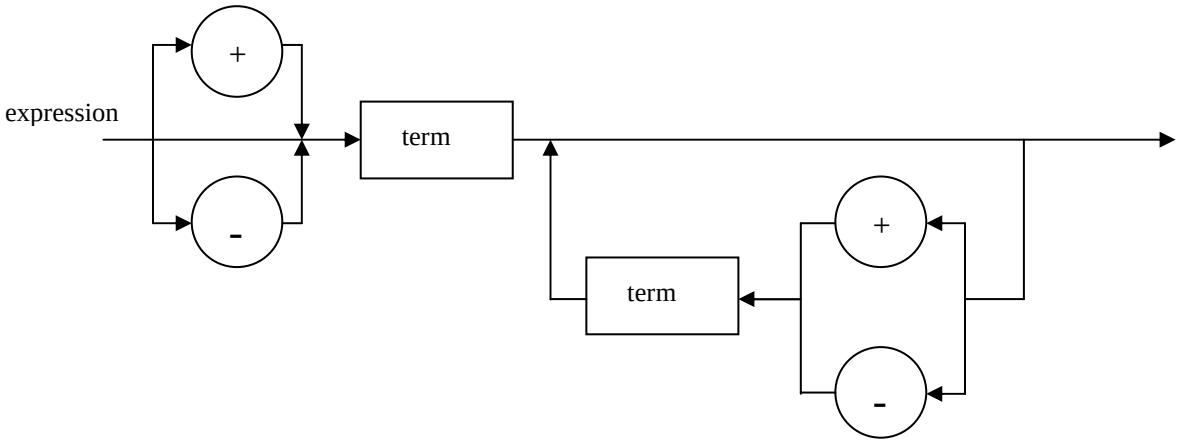
```
program SYNTAKTICKA_ANALYZA
definice vyjmenovaneho typu SYMBOL = (IDENT, PRIRAZ, PLUS, KRAT,  LEVA,
                                                    PRAVA, IFS, THENS, ELSES);
 promenna  N typu SYMBOL;
 procedura CTI(vystupni parametr S typu SYMBOL)   ...
 procedura CHYBA(vstupni parametr CISLO typu integer)  ...
 procedura TISK(vstupni parametr CISLO typu integer)  ...

procedura S
{ if N =  IFS then
              { TISK(2); CTI(N); E;
                  if N ≠ THENS then CHYBA(2);
                  else { CTI(N); S; Z;
                         }
              }
    else
      { TISK(1); V; if N ≠ PRIRAZ then CHYBA(1);
                       else { CTI(N); E;
                              }
      }   /* vstupni řetězec patří do jazyka */
}

 procedura Z
{ if N = ELSES then { TISK(3); CTI(N); S;
                       }
  else TISK(4);         /*bude se chovat tak, jak jsme nakreslili strom, nebo jinak? */
}

 procedura E
{ TISK(5); T;
     while N = PLUS do {  CTI(N); T;
                          }
}
```

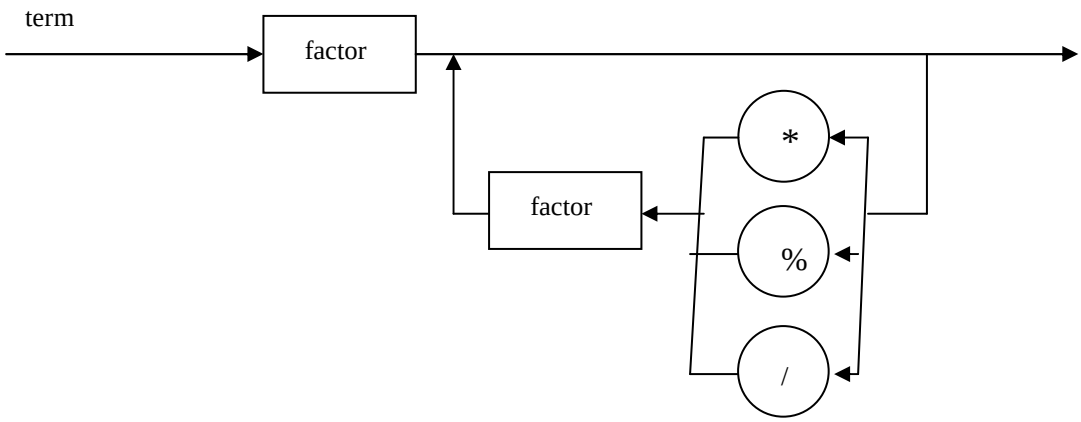
```
procedura T
{ TISK(6); F;
     while N = KRAT do {  CTI(N); F;
                          }
}

procedura F
{ if N = LEVA then { TISK(7); CTI(N); E;
                          if N ≠ PRAVA then CHYBA(7)
                          else CTI(N)
                   }
  else { TISK(8); V; }
}

procedura V
{ if N ≠ IDENT then CHYBA(9)  else { TISK(9);CTI(N);I;
                                     }
}

procedura I
{ if N = LEVA then { TISK(10); CTI(N); E;
                          if N ≠ PRAVA then CHYBA(10)
                                       else CTI(N)
                   }
                 else TISK(11);
}

procedura  MAIN
{  CTI(N); S;
}
```

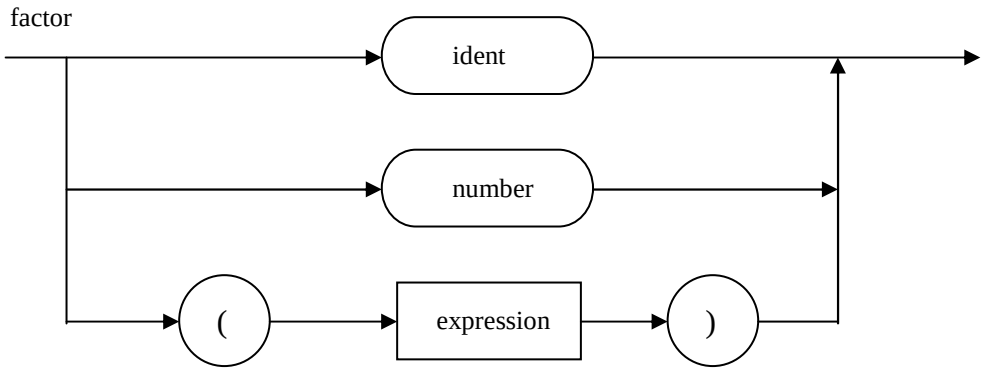
Syntax jazyka PL0




condition
ODD
expression
expression
=
#
<
<=
>
>=
expression


+
expression
term
-
+
term
-


term
factor
*
factor
%
/


factor
ident
number
(
expression
)

X	FIRST (X)	FOLLOW (X)
BLOCK	const , var , procedure ident , call , begin if , while , ε	. , ;
STATEMENT	ident , call , begin if , while , ε	. , ; , end
CONDITION	odd , + , − , ident number , (	then , do
EXPRESSION	(, ident , number , + , −	) , . , ; , = # , < , <= , > , >= then , do , end
TERM	(, ident , number	) , . , ; , = # , < , <= , > , >= then , do , end + , −
FACTOR	(, ident , number	) , . , ; , = # , < , <= , > , >= then , do , end + , − , * , /, %

```
/*C program Lexikální a Syntaktická analýza PL0 – upozorníme jen na hlavní části */

#define NSYM 35              /* pocet rozpoznatelnych symbolu*/
#define NORW 11             /* pocet klicovych slov */
#define TMAX 100            /* velikost tabulky identifikatoru */
#define NMAX 5             /* maximalni pocet cislic v cisle */
#define AL 10             /* delka identifikatoru */
#define CHSETSIZE 128       /* pocet znaku v mnozine */
#define MAXERR 30           /* maximalni pocet chyb */
#define AMAX 2048           /* nejvyssi adresa */
#define LEVMAX 3            /* maximalni hloubka vnoreni */
#define CXMAX 200           /* velikost prostoru pro kod */
#define STACKSIZE 500   /* vypoctovy zásobník*/
```

definice konstant překladače

```
typedef enum {null, ident, number, plus, minus, times, slash, modulo, oddsym, eql,
        neq, lss, leq, gtr, geq, lparen, rparen, comma, semicolon, period,
        becomes, beginsym, endsym, ifsym, thensym, whilesym, dosym, callsym,
        constsym, varsym, procsym} SYMBOL;
typedef char ALFA[AL]; /*pole  k ulozeni textu identifikatoru*/
typedef enum {constant, variable, procedure} OOBJECT; /*druh identifikatoru*/
typedef int SYMSET[NSYM];
```

lex.symboly

```
char ch;                /*posledni precteny znak*/
FILE *iva, *zdroj;      /*pomocny soubor pro vypis generovaneho kodu*/
                /*a tabulky symbolu*/

int txpom;              /*pomocna promenna*/
SYMBOL sym;             /*posledni precteny symbol*/
ALFA id;                /*posledni precteny identifikator*/
int num;                /*posledni prectene cislo*/
int cc;                 /*pocet znaku*/
int ll;                 /*delka radku*/
int kk,err;
int cx;                 /*pocitadlo adres; vice bude receno v kap. Pridelovani pameti */
char line[81];          /*nacteny radek*/
ALFA a;
ALFA word[NORW]={"begin", "call", "const", "do", "end", "if", "odd", "procedure",
                 "then", "var", "while"};          /*pole rezervovaných identifikátorů*/
SYMBOL wsym[NORW]={beginsym, callsym, constsym, dosym, endsym, ifsym, oddsym,
                   procsym, thensym, varsym, whilesym}; /*poradi převede na vyctov.typ*/
SYMBOL ssym[255]; /*+,-,*,…, jednoznakove oddělovače, plneno v main*/
SYMSET declbegsys, statbegsys, facbegsys;

struct {
   ALFA name;          /*jmeno*/
   OOBJECT kind;       /*druh*/
   union {
      int val;
      struct {
         int level,adr,size;
      } vp;
   } CO;
} TABLE[TMAX+1];
```

atributy identifikatoru

Tabulka symbolů

```
/* nacita ze vstupniho souboru 1 znak do glob. promenne 'ch' a prekroci konec radky */

void getch(void) {   /*sklada znaky do pole line*/
   if (cc == ll) {
```

```
      if (feof(zdroj)) {
         printf("program incompleted");
         exit(2);
      }
      ll = cc = 0;
      printf("      ");

      do {
         fscanf(zdroj,"%c",&ch);
         if ((ch != '\n') && (ch != '\r')) {
            line[ll++]=ch;                    /*pridani znaku do promenne line */
            printf("%c",ch);
         }
      } while ((ch != '\n') && (ch != '\r') && (feof(zdroj) == 0));

      printf("\n");
      line[ll++] = ' ';
   }
   ch = line[cc++];
}   // getch()

/* Lex.analyza nacita ze vstupniho souboru 1 symbol a vrati jeho kod do glob. promenne 'sym' */
void getsym(void) {
int i, j, k;

   while (ch <= ' ') getch();   /* netisknutelne znaky */
   if ((ch >= 'a') && (ch <= 'z')) {   /* identifier or reserved word */
      k = 0;

      do {
         if (k < AL) a[k++] = ch;
         getch();
      } while (((ch >= 'a') && (ch <= 'z')) || ((ch >= '0') && (ch <= '9')));

      a[k] = '\0';
      strcpy(id, a);
      i = 0;
      j = NORW - 1;

      do {
         k = (i + j) / 2;
         if (strcmp(id, word[k]) <= 0) j = k - 1;

         if (strcmp(id, word[k]) >= 0) i = k + 1;
      } while (i <= j);

      if ((i - 1) > j) sym = wsym[k];
      else sym = ident;
   }
   else
      if ((ch >= '0') && (ch <= '9')) {   /* number */
         k = num = 0;
         sym = number;

         /* vypusti pripadne pocatecni nuly u cisla */
        /* while (ch == '0') getch(); */

         do {
            num = 10 * num + (ch - '0');  /*vypocet hodnoty cisla */
```

*/*pulenim intervalu hleda v poli word*/*
*/*zda to je rezervovany identifikátor */*

```
            k++;
            getch();
         } while ((ch >= '0') && (ch <= '9'));
         if (k > NMAX) error(30);
      }
      else
         if (ch == ':') {
            getch();
            if (ch == '=') {                    prirazeni
               sym = becomes;
               getch();
            }
            else sym = null;
         }
         else
            if (ch == '<') {
               getch();
               if (ch == '=') {                 /*mensi roven*/
                  sym = leq;
                  getch();
               }
               else
                  if (ch == '>') {
                     sym = neq;                 /*neroven*/
                     getch();
                  }
                  else
                     sym = lss;                 /*mensi*/
            }
            else
               if (ch == '>') {
                  getch();
                  if (ch == '=') {              /*vetsi roven*/
                     sym = geq;
                     getch();
                  }
                  else sym = gtr;
               }
               else {
                  sym = ssym[ch];               /*jednoznakovy oddělovač viz main*/
                  getch();
               }
} /* konec procedury getsym */
```

```
/* vlozi object do tabulky symbolu
      k    :typ objektu, tj. zda jde o konstantu,promennou,...
      lev :uroven, ve ktere je objekt deklarovan
      dx  :adresa objektu
*/
void enter(OOBJECT k,int *tx,int lev,int *dx) {

   (*tx)++;       /* inkrementuje index  tabulky symbolu */
   txpom = *tx; /* pro vypis TS */
   strcpy(TABLE[*tx].name,id);
   TABLE[*tx].kind=k;

   switch (k) {
      case constant: if (num>AMAX) {
                                 error(31);
                                 num = 0;
                     }
                     TABLE[*tx].CO.val = num;
                     break;

      case variable: TABLE[*tx].CO.vp.level = lev;
                                 TABLE[*tx].CO.vp.adr = (*dx)++;
                                 break;

      case procedure: TABLE[*tx].CO.vp.level = lev;
                                  break;

   }
}   // enter()
```

/*plneni tab. symbolu*/

```
/* vyhleda symbol v tabulce symbolu
      id   :jmeno symbolu
      tx   :ukazovatko na konec tabulky symbolu
   navratova hodnota:
      -1 : symbol nenalezen
     >=0 : adresa symbolu
*/
int position(ALFA id,int tx) {
int i;

   strcpy(TABLE[0].name,id);          /*sentinel*/
   i = tx;                            /*hleda od posledního zarazeneho (respektuje lokalitu*/
   while (strcmp(TABLE[i].name,id)) i--;

return(i);
}   // position()
```

```
/* zpracovani deklarace konstanty ve tvaru:
       ident = hodnota.
       tx   :ukazovatko na volne misto v tabulce symbolu
       lev :uroven, ve ktere je symbol deklarovan
       dx  :adresa - neni pouzita, protoze jde tady o kontantu
*/
void constdeclaration(int *tx,int lev,int *dx) {

if (sym == ident) {
    getsym();
    if ((sym == eql) || (sym == becomes)) {
        if (sym == becomes) error(1);  /*v deklaraci konstant musí byt „=“  */
        getsym();
        if (sym == number) {
            enter(constant,tx,lev,dx);   /*ulozeni konstanty do Tab.Symb, */
            getsym();
        } else error(2);      /*konstante není prirazeno cislo*/
    } else error(3);          /*nenasel = ani prirazeni*/
} else error(4);              /*nenasel identifikátor*/
}   // constdeclaration()

/* zpracovani deklarace promenne
       tx   :ukazovatko na volne misto v tabulce symbolu
       lev :uroven,ve ktere je symbol deklarovan
       dx  :adresa promenne
*/
void vardeclaration(int *tx,int lev,int *dx) {

if (sym == ident) {
    enter(variable,tx,lev,dx);
    getsym();
} else error(4);

}   // vardeclaration()
```

```
void factor(…) {    /*v kompletnim tvaru bude mit parametry*/
int i;

      while (facbegsys[sym]) {   /* facbegsys se naplni v main*/
      if (sym == ident) {
         i = position(id,tx);
         if (i == 0) error(11);      /*nenalezen v Tab.Symb.*/
         else
         getsym();
      } else
         if (sym == number) {
            if (num > AMAX) {
               error(31);
               num = 0;
            }
            getsym();
         } else
            if (sym == lparen) {                  getsym();
               expression(…);
               if (sym == rparen) getsym();
               else error(22);
            }
   }
}   // factor()


void term(…) {           /*v kompletnim tvaru bude mit parametry*/

SYMBOL mulop;
   factor(…);

   while ((sym == times) || (sym == slash) || (sym == modulo)) {
      mulop = sym;
      getsym();
      factor(…);
   }

}   // term()
```

```
void expression(…) {   /*v kompletnim tvaru bude mit parametry*/
SYMBOL addop;

   if ((sym == plus) || (sym == minus)) {  /*unární operatory*/
      getsym();
      term(…);
   }
   else {
      term(…);
   }

   while ((sym == plus) || (sym == minus)) { /*binární operatory*/
      getsym();
      term(…);
   }
}   // expression()


void condition(…) {
SYMBOL relop;

   if (sym == oddsym) {
      getsym();
      expression(…);
   }
   else {
      expression(…);
      if ((sym != eql) && (sym != neq) && (sym != lss) && (sym != gtr) && (sym != leq) &&
         (sym != geq))
         error(22);
      else {
         getsym();
         expression(…);
          }
   }

}   // condition()


void statement(…) {    /*v kompletnim tvaru bude mit parametry*/
int i;

   if  (sym != ident) {
      error(10);
      do
         getsym();
      while (fsys[sym] == 0);
   }
   if (sym == ident) { /*nalezen prikaz prirazeni*/
      i = position(id,tx);
      if (i == 0) error(11);         /*nenasel se identifikátor*/
      else
         if (TABLE[i].kind!=variable) { /*prirazeni do jineho ident. nez promenna*/
            error(12);
            i = 0;
         }
      getsym();
      if (sym == becomes) getsym();
      else error(13);
```

```
        expression(…);
    }
    else
        if (sym == callsym) {/*nalezeno volani podprogramu*/
            getsym();
            if (sym != ident) error(14);
                  else {
                     if ((i = position(id,tx)) == 0) error(11);
                     else {
                        if (TABLE[i].kind == procedure) gen(cal,  …
                                                 else error(15);
               }
                     getsym();
                  }
        }
        else
            if (sym == ifsym) { /*podmineny prikaz*/
               getsym();
               condition(…);

               if (sym == thensym) getsym();
               else error(16);
               statement(…);
            }
            else
               if (sym == beginsym) { /*zacina novy blok*/
                  getsym();
                  statement(…);

                  while  (sym == semicolon) {
                     getsym();
                     else error(10);
                     statement(…);
                  }

                  if (sym == endsym) getsym(); /*konci predchozi blok*/
                  else error(17);
               }
               else
                  if (sym == whilesym) { /*zacina cyklus while*/
                     condition(…);
                     if (sym == dosym) getsym();
                     else error(18);
                     statement(…);
                  }

}   // statement()
```

```
void block(…) {          /*v kompletnim tvaru bude mit parametry*/
   do {
      if (sym == constsym) { /*deklaracni cast konstant*/
         getsym();
         do {
            constdeclaration(…);
            while (sym == comma) {
               getsym();
               constdeclaration(…);
            }
            if (sym == semicolon) getsym();
            else error(5);
         } while (sym==ident);
      }

      if (sym == varsym) { /*deklaracni cast promennych*/
         getsym();
         vardeclaration(…);

         while (sym == comma) {
            getsym();
            vardeclaration(…);
         }
         if (sym == semicolon) getsym();
                  else error(5);
      }

   while (sym == procsym) { /*definice podprogramu*/
         getsym();
         if (sym == ident) {
            enter(procedure);
            getsym();
         } else error(4);
         if (sym==semicolon) getsym();
         else error(5);
         block(…);
         if (sym == semicolon) {
            getsym();
         } else error(5);
      }
   } while (declbegsys[sym]);  /* declbegsys se plni v main*/

   statement(…);

}   // block()
```

```
/*hlavni program*/
main(void) {
char zdrojak[13];

   /* cte jmeno souboru, dokud uzivatel nezada nenulovy retezec */
   do {
      printf("Zadej jmeno souboru obsahujiciho zdrojovy text: ");
      scanf("%s",zdrojak);
   } while (strlen(zdrojak) < 1);

   if ((iva = fopen("TAB.SYM", "w")) == NULL) {
      printf("\nCHYBA! Nepodarilo se otevrit soubor pro zapis tabulky symbolu...\n");
      return(-1);
   }
   /* ...a pak otestuje, jestli soubor existuje */
   if ((zdroj = fopen(zdrojak, "r")) == NULL) {
      printf("\nCHYBA! Nepodarilo se otevrit soubor se zdrojovym textem [%s]...\n",zdrojak);
      return(-1);
   }
   for (ch=' ';ch<='_';ch++) ssym[ch] = null;
   ssym['+'] = plus;
   ssym['-'] = minus;
   ssym['*'] = times;
   ssym['/'] = slash;
   ssym['%'] = modulo;
   ssym['('] = lparen;
   ssym[')'] = rparen;
   ssym['='] = eql;          /*naplneni hodnot jednoznakových oddelovacu*/
   ssym[','] = comma;
   ssym['.'] = period;
   ssym['#'] = neq;
   ssym['<'] = lss;
   ssym['>'] = gtr;
   ssym[';'] = semicolon;
   nuluj(declbegsys);
   nuluj(statbegsys);
   nuluj(facbegsys);
   /*v deklaracni casti se musi zacinat bud 'const','var' nebo 'procedure'*/
   declbegsys[constsym] = declbegsys[varsym] = declbegsys[procsym] = 1;
   /*ve statementu se musi zacinat bud 'begin','call','if','while' nebo ident.*/
   statbegsys[beginsym] = statbegsys[callsym] = statbegsys[ifsym] = statbegsys[whilesym] = 1;
   /*faktor muze byt bud ident., cislo nebo leva zavorka*/
   facbegsys[ident] = facbegsys[number] = facbegsys[lparen] = 1;
   ch = ' ';
   kk = AL;
   getsym();
   block(…); /*zavola preklad programu*/
   if (sym != period) error(9); /*a konci teckou*/
   listtabsym();
   if (err == 0) {
      printf("\nno error in PL/0 program\n");
   }
   else printf("\n %2d error(s) in PL/0 program",err);
return(0);
}
```

Zpracování chyb v PL0

Panický způsob zotavování – triviální strategie:
vynechá text až do místa, kde se snadno vzpamatuje. Snadno se vzpamatuje v místě s významným symbolem.

Předpoklady:

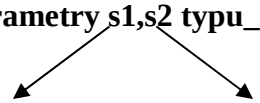
1. **Každý typ příkazu začíná jiným symbolem.**
2. **„ „ deklarace „ „ „ .**
3. **Každá vyvolaná procedura se provede až do konce (žádný chybový výstup).**

Zásady:

1. **Každá procedura má parametr – množinu následujících symbolů.**
2. **Při chybě je přeskočen vstupní text až k legálně následujícímu symbolu za prováděnou procedurou.**
3. **Na konci procedury je proveden Test, který ověří, že příští symbol patří do množiny následovníků.**
4. **Pro zmenšení vynechávaných úseků se do následovníků přidávají symboly ze začátku důležitých konstrukcí (tzv. STOP SYMBOLY). Zdůvodněte si to.**

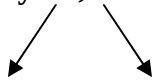
Činnost zajišťuje procedura

Test(parametry s1,s2 typu_množina_symbolů; n celočíselné_označení_chyby)



Následující symboly **Stop symboly**

Procedura Test je využitelná k ověření akceptovatelnosti symbolů na začátku procedur SA. Symboly s1 , s2 mají pak jiný význam



Počáteční **Následující**

Seznam chyb

1	**pouzito "=" misto ":="**
2	**za "=" musi nasledovat cislo**
3	**za identifikatorem ma nasledovat "="**
4	**za "const", "var", "procedure" musi nasledovat identifikator**
5	**chybi strednik nebo carka**
6	**nespravny symbol po deklaraci procedury**
7	**je ocekavan prikaz**
8	**neocekavany symbol za prikazovou casti bloku**
9	**ocekavam tecku**
10	**nespravny symbol v prikazu**
11	**nedeklarovany identifikator**
12	**prirazeni konstante a procedure neni dovoleno**
13	**operator prirazeni je ":="**
14	**za "call" musi nasledovat identifikator**
15	**volani konstanty nebo promenne neni dovoleno**
16	**ocekavano "then"**
17	**ocekavano "}" nebo ";"**
18	**ocekavano "do"**
19	**nespravne pouzity symbol za prikazem**
20	**ocekavam relaci**
21	**jmeno procedury nelze pouzit ve vyrazu**
22	**chybi uzaviraci zavorka**
23	**faktor nesmi koncit timto symbolem**
24	**vyraz nesmi zacinat timto symbolem**
30	**prilis velke cislo**

Alternativa pro C++

```
/* testovat zda nacteny symbol je v mnozine symbolu 's1'.
   Pokud neni generuje chybu a nacita opakovane ze vstupu
   dokud neni nacten symbol z mnozin 's1' a 's2'
*/
void test(SYMSET s1,SYMSET s2,int n) {
SYMSET pom;

   if (!s1[sym]) {        /*sym není v množině s1 */
      error(n);
      nuluj(pom);
      sjednot(pom,s1);    } /*sjednoti s1, s2 do pom */
      sjednot(pom,s2);

      while (pom[sym] == 0) getsym();  /* pokud sym není v s1∪ s2 cti další */
   }
}   // test()

void expression(SYMSET fsys) {
SYMBOL addop;
SYMSET pom;

   if ((sym == plus) || (sym == minus)) { /*unární plus, minus  */
      addop = sym;
      getsym();
      nuluj(pom);
      sjednot(pom,fsys);
      pom[plus] = pom[minus] = 1;
      term(pom);       /*volame term(fsys ∪ plus ∪ minus ) */
    }
   else {
      nuluj(pom);
      sjednot(pom,fsys);
      pom[plus] = pom[minus] = 1;   /*bez unárního plus minus */
      term(pom);
   }

   while ((sym == plus) || (sym == minus)) {
      addop = sym;
      getsym();
      nuluj(pom);                          /* iterace {+T}  */
      sjednot(pom,fsys);
      pom[plus] = pom[minus] = 1;
      term(pom);

    }
}   // expression()
```

```
void term(SYMSET fsys) {
SYMBOL mulop;
SYMSET pom;

   nuluj(pom);
   sjednot(pom,fsys);
   pom[times] = pom[slash] = pom[modulo] = 1;
   factor(pom);        /*volame factor( followE ∪  follow T */

   while ((sym == times) || (sym == slash) || (sym == modulo)) {
      mulop = sym;
      getsym();
      sjednot(pom,fsys);
      pom[times] = pom[slash] = pom[modulo] = 1;
      factor(pom);
   }
}  // term()

void factor(SYMSET fsys) {
int i;
SYMSET pom;

   test(facbegsys,fsys,24);            /* test na zacatku faktoru */
   while (facbegsys[sym]) {
      if (sym == ident) {
         i = position(id,tx);
         if (i == 0) error(11);        /* ten identifikátor není deklarovany */
         getsym();
      } else
         if (sym == number) {
            if (num > AMAX) {
               error(31);
               num = 0;
            }
            getsym();
         } else
            if (sym == lparen) {
               getsym();
               nuluj(pom);
               sjednot(pom,fsys);
               pom[rparen] = 1;
               expression(pom);
               if (sym == rparen) getsym();/* follow „lparen expression“ je rparen */
               else error(22);             /* paren chybi */
            }
            nuluj(pom);
            pom[lparen] = 1;         /* pokud ses nezotavil drive, preskoc jen k lparen */
            test(fsys,pom,23);       /* test na konci faktoru */
   }
}  // factor()
```

Sémantické zpracování

(při Synt.Anal. prováděné rekurzivním sestupem)

Atributy = vlastnosti gramatických symbolů nesoucí sémantickou informaci (hodnota, adresa, typ, apod.)

Sémantické zpracování zahrnuje vyhodnocení atributů symbolů v derivačním stromu

Způsoby vyhodnocení (pro praxi je zajímavé jednoprůchodové vyhodnocování) :

1. **procházením stromem od listů ke kořenu se vypočtou tzv. syntetizované atributy**
2. **„ „ „ od rodiče k potomkovi , od staršího bratra k mladšímu se vypočtou tzv. dědičné atributy**

Rekurzivním sestupem řízený překladač provádí výpočet atributů v čase zpracování pravých stran pravidel formou:

- **Zpracování neterminálu ≡ vyvolání procedury ≡ expanze neterminálu při derivaci,**
- **srovnávání terminálů (ze vstupu a z pravé strany pravidla)**

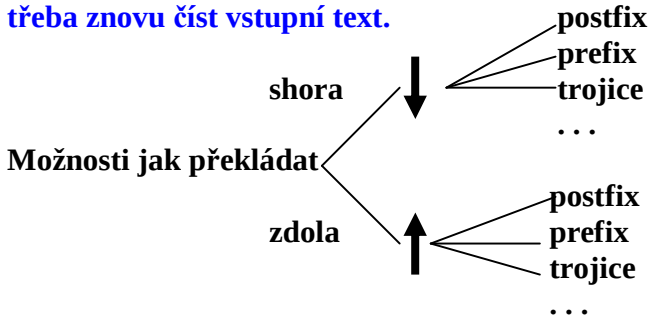
Nutno doplnit procedury LA i SA takto:

-LA bude předávat s přečteným vstupním symbolem i jeho atributy

-procedury SA pro neterminály doplnit o:

- **parametry vstupní, odpovídající dědičným atributům,**
- **parametry výstupní odpovídající syntetizovaným atributům,**
- **zavést lokální proměnné pro uložení atributů pravostranných symbolů,**
- **před vyvoláním procedury korespondující neterminálu z pravé strany vypočítat hodnoty jeho dědičných atributů,**
- **na konec procedury popisující pravou stranu pravidla zařadit příkazy vyhodnocující syntetizované atributy levostranného symbolu.**

Snažíme se vymyslet to tak, aby se vyhodnocení provedlo jedním průchodem a nebylo třeba znovu číst vstupní text.



Překlad příkazů

Překlad přiřazovacího příkazu do čtveřic shora dolů (rekurzivním sestupem)

Gramatika: **S --> a := E ;**
E --> T { + T }
T --> F { * F)
F --> (E) | a

Atributy symbolů E, T, F jsou:
PA = počáteční adresa pro mezivýsledky
KA = koncová adresa (již nezabraná)
AH = adresa hodnoty

Přiřazení atributů symbolům:

	Atributy	Symboly které ho mají
Dědičný	**PA**	**S, E, T, F**
Syntetizovaný	**AH**	**a, E, T, F**
Syntetizovaný	**KA**	**S, E, T, F**

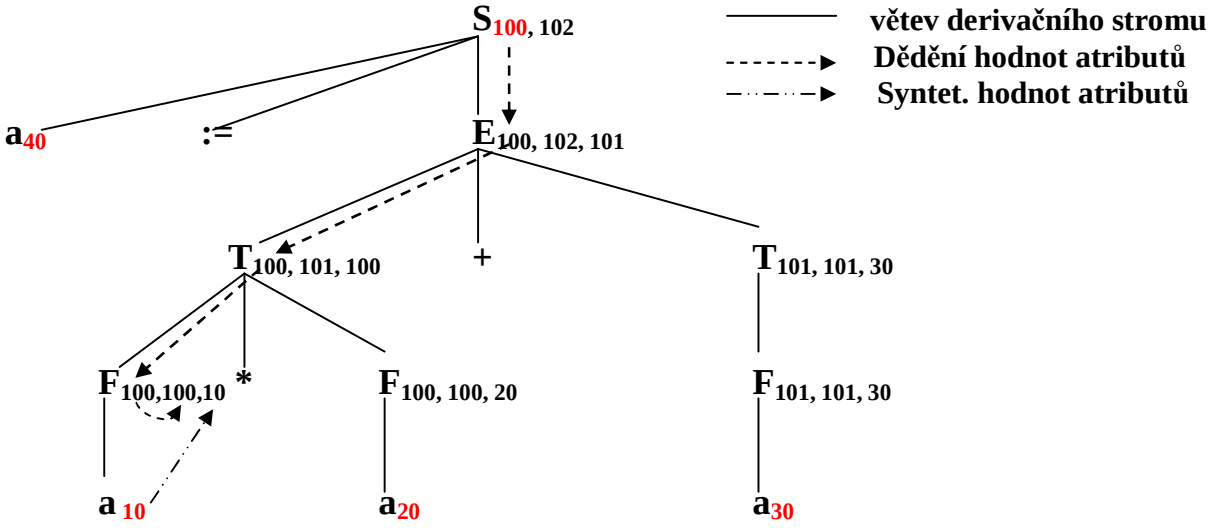
Budeme generovat čtveřice tvaru: **(+, adr1, adr2, adr3)**
(s významem (operátor, operand, operand, výsledek) **(*, adr1, adr2, adr3)**
(=, adr1, adr2, 0)

Vstupem jsou symboly zjištěné LA spolu s jejich atributy
Výstupem je posloupnost instrukcí vnitřního jazyka (atributy jsou jejich součástí)
Mezivýsledky budeme v příkladu ukládat od adresy 100

Např. $a_{40} = a_{10} * a_{20} + a_{30}$ **přeloží se na**

***, 10, 20, 100**
+, 100, 30, 101
=, 101, 40, -

Indexy znamenají adresu
Atributy uzlů stromu jsou v pořadí PA, KA, AH, lze je vyhodnotit procházením stromem.



Symbolický zápis programu překladače přiřazovacího příkazu do čtveřic

```
program PREKLAD          bude používat tyto proměnné a podprogramy:

   typ SYMBOL = záznam TYP:(IDENT, PRIRAZ, PLUS, KRAT,
                                   STREDNIK, LEVA, PRAVA);
                               ADRESA: INTEGER;
                      konec záznamu;
procedura CTI(S) /*cte lexikální symbol do promenne S typu SYMBOL*/
procedura CHYBA ...
procedura VYSTUP(… /*tiskne instrukce tvaru řetězec, adresa, adresa, adresa*/

promenna N: SYMBOL;

procedura S  {
   promenna L: SYMBOL; KA, AH: integer;
     CTI(L);
     if L.TYP <> IDENT then CHYBA;
     CTI(N);
     if N.TYP <> PRIRAZ then CHYBA;
     CTI(N);
     E(100, KA, AH); /*před vyvoláním procedury se určí hodnota dědičných atributů */
     if N.TYP <> STREDNIK then CHYBA;
     VYSTUP('=', AH, L.ADRESA, 0);
}                              vstupní param.        výstupní parametry

procedura E(hodnota PA: integer; adresa KA, AH: integer) {
   promenna AH2: integer;
   T(PA, KA, AH);
     while N.TYP = PLUS DO
        { CTI(N);
            T(KA, KA, AH2);
            VYSTUP('+', AH, AH2, KA);
            AH ← KA;
            KA ← KA + 1;
        }
}
```

```
procedura T(hodnota PA: integer; adresa KA, AH: integer) {
  promenna AH2: integer;
  F(PA, KA, AH);
    while N.TYP = KRAT do
      { CTI(N);
          F(KA, KA, AH2);
          VYSTUP('*', AH, AH2, KA);
          AH ← KA;
          KA ← KA + 1;
      }
}

procedura F(hodnota PA: integer; adresa KA, AH: integer); {
 if N.typ = IDENT then
      { AH ← N.ADRESA;
          CTI(N);
          KA ← PA;
      }
      else if N.TYP <> LEVA then CHYBA
          else { CTI(N);
                   E(PA, KA, AH);
                   if N.TYP <> PRAVA then CHYBA
                              else CTI(N);
               }
}

procedura Main  {
      S;
}
```

uvažujme příklad	**a**	**:=**	**a**	**+**	**a**
uložení na adr.	**10**		**20**		**30**
použijeme označení	**a(10)**		**a(20)**		**a(30)**

výsledkem překladu budou čtveřice: **(+,20,30,100)**
(=, 100, 10, 0)

viz následující obrázek

S	E	T	F
čte a(10), :=, **a(20)** **tj. L <- a(10)** **N <- a(20)** **volá** **E(100,KA,AH)**			
	volá **T(100,KA,AH)**		
		volá **F(100,KA,AH)**	
			AH <- 20 **čtením** **N <- PLUS** **KA <- 100**
		návratem **AH <- 20** **KA <- 100**	
	návratem **AH <- 20** **KA <- 100** **čtením** **N <- a(30)** **volá** **T(100,KA,AH2)**		
		volá **F(100,KA,AH)**	
			AH <- 30 **čtením** **N <-STREDNIK** **KA <- 100**
		návratem **AH <- 30** **KA <- 100**	
	návratem **AH2 <- 30** **KA <- 100** **výstup** **(+,20,30,100)** **AH <- 100** **KA <- 101**		
návratem **AH <- 100** **KA <- 101** **výstup** **(=, 100, 10, 0)**			

<u>Rozšíření jazyka na seznam příkazů a příkaz if</u>. < > označují neterminální symbol

G[<SP>] :

<SP> --> S ; <SP>

<SP> --> e

S --> a := E

S --> if <BE> then S

zde budeme generovat instrukci pro přeskok S při <BE> false, ve tvaru JIF, AH, misto_za_S. Nevíme zatím ale kam, tak si číslo instrukce zapamatujeme a dodatečně pak kompletujeme

Vygeneruje čtveřici, která uloží do AH hodnotu booleovského výrazu

pracuje také s následujícími proměnnými a podprogramy:

```
promenna      N: SYMBOL;
              C, KA: integer;   /* C bude citac instrukci, tj. radek výstupního kodu */
 procedura DOPLN(hodnota KAM, CO: integer) ... /*kompletuje čtveřici*/   {
   /*...  není rozepsana */ }
 procedura S(hodnota PA: integer; adresa KA: integer) {
   promenna L: SYMBOL;
      AH, KA1, PC: integer;   /* PC slouzi pro zapamatovani citace instrukci C */
  if N.TYP = IFS then
      { CTI(N);
          BE(PA, KA, AH);
          VYSTUP('JIF', AH, 0, 0);
          PC ← C; /* C je čítač instrukcí, inkrementovaný procedurou VYSTUP */
          if N.TYP <> THENS then CHYBA;  CTI(N);
          S(KA, KA1); //paměť obsazuje od KA, KA1 je již neobsazené místo
          DOPLN(PC-1, C);
          KA ← KA1;
      }
    else
      { if N.TYP <> IDENT then CHYBA;
          L ← N;
          CTI(N); if N.TYP <> PRIRAZ then CHYBA;   /*prirazovaci prikaz */
          CTI(N); E(PA, KA, AH);
          VYSTUP(':=', AH, L.ADRESA, 0);
      }
}

 procedura SP(hodnota PA: integer; adresa KA: integer)  {
   promenna KA1: integer;
  if N <> PRAZDNY_SYMBOL then       /*prázdným symbolem bude eof */
      { S(PA, KA);
          if N <> STREDNIK then CHYBA;
          CTI(N);  SP(KA, KA1); KA ← KA1;
      }
}
```

```
 procedura BE(hodnota PA: integer; adresa KA, AH: integer) ...
    /*je podobná E, pracuje s or, and, not, <, >, =, ... */
/* další procedury  pro E, T, F, VYSTUP   */

procedura MAIN  {
      C ← 1; (* C je dale inkrementováno procedurou VYSTUP *)
      CTI(N);
      SP(100, KA);  /* pomocne promenne pro mezivysledky zacinaji na adr 100 */
 }
```

Např.

C	:=	A	*	B	;	if	A	<	B	then	C	:=	A	+	B	;
30		10		20			10		20		30		10		20	

se přeloží na:

(1) '*' , 10, 20, 100

(2) '=' , 100, 30, 0

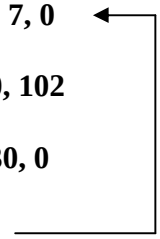
(3) '<' , 10, 20, 101

(4) 'JIF', 101, 7, 0

(5) '+' , 10, 20, 102

(6) '=' , 102, 30, 0

(7)



procedura DOPLN(7, 4) doplni adresu 7

Překlad posloupnosti příkazů do postfixové notace

Předpokládejme gramatiku:

<SP> --> S ; <SP> ***/*nepotřebuje atributy kromě adr identifikátorů*/***

<SP> --> e

S --> a := E

S --> if E then S **/*true/false je vyjádřitelné ≠0 / =0 */**

E --> T { + T }

T --> F { * F }

F --> (E)

F --> a

Budeme používat postfixové instrukce:

PLUS vybere ze zásobníku dvě hodnoty, sečte je a
výsledek uloží do zásobníku

KRAT ---- " ---- vynásobí je a
výsledek uloží do zásobníku

TA adr vloží do zásobníku adresu adr

DR dereferencuje vrchol zásobníku (nahradí adresu jejím obsahem)

ST uloží hodnotu z vrcholu zásobníku na adresu
uloženou pod vrcholem zásobníku

IFJ m je-li vrchol zásobníku false, provede skok na
m - tou postfixovou instrukci

kromě adresy identifikátoru nepotřebujeme žádné další atributy

Programový zápis používá následující proměnné a podprogramy

```
promenna   N: SYMBOL;      /*doda lexikální analyzátor*/
           C; integer;     /*čítač umístění další generované instrukce*/
   ...
procedura S  {
 promenna PC: integer;     /*pomocný čítač pro zapamatování C*/
if N.TYP = IDENT then
      { VYSTUP('TA', N.ADRESA);
        CTI(N);
        if N.TYP <> PRIRAZ then CHYBA;
        CTI(N);
        E;
        VYSTUP('ST', 0);  /*druhý parametr je nevýznamový*/
      }
     else
      { if N.TYP <> IFS then CHYBA;
          else
            { CTI(N);
              E;
              PC ← C;
              if N.TYP <> THENS then CHYBA;
              VYSTUP('IFJ', 0);
              CTI(N);
              S;
              DOPLN(PC, C);
            }
      }
}
```

```
procedura SP   {
    if N.TYP <> PRAZDNY_SYMBOL then
        {       S;
                if N.TYP <> STREDNIK then CHYBA;
                CTI(N);
                 SP;
        }
}

procedura E   {
    T;
     while N.TYP = PLUS do
        {       CTI(N);
                T;
                VYSTUP('PLUS', 0) /*0 znamená, že u PLUS je parametr nevýznamový*/
        }
}
procedura T   {
     F;
     while N.TYP = KRAT do
        {       CTI(N);
                 F;
                VYSTUP('KRAT', 0);
        }
}
```

```
procedura F   {
  if N.TYP = IDENT then
       { VYSTUP(TA, N.ADR);
           VYSTUP(DR, 0);
           CTI(N);
       }
      else    if N.TYP <> LEVA then CHYBA
                 else  { CTI(N);
                           E;
                           if N.TYP <> PRAVA then CHYBA
                                       else CTI(N);
                       }
}
```

```
procedura MAIN
{
       C ←1;
       CTI(N);
       SP;
}
```

Např. $a_{10} := a_{20} + a_{30}$ **se přeloží na**

```
TA  10
TA 20
DR
TA 30
DR
PLUS
ST
```

Sémantické zpracování v PL0

Instrukce postfixového zápisu

lit 0,A :uloz konstantu A do zasobniku

opr 0,A :proved instrukci A

1	unarni minus
2	+
3	-
4	*
5	div
6	mod
7	odd
8	=
9	<>
10	<
11	>=
12	>
13	<=

lod L,A :ulož hodnotu proměnné z adr. L,A na vrchol zásobníku

sto L,A :zapiš do proměnné s adr. L,A hodnotu z vrcholu zásob.

cal L,A :volej proceduru s adresou kódu A z úrovně L

ret 0,0 :return, v některých verzích opr 0,0

ing 0,A :zvyš obsah Top-registru zásobníku o hodnotu A
v některých verzích int 0,A

jmp 0,A :proveď skok na adresu kódu A

jpc 0,A :proveď podmíněný skok na adresu kódu A

Následující kód je lepší si prohlédnout z obrazovky, než-li jej tisknout. Části týkající se překládání jsou vybarvené.

```
/*postfixove instrukce jsou ve tvaru*/
typedef struct {
   FCT f;    /*kod instrukce = postfixový operátor=funkce instrukce*/
   int l;    /*uroven vnoreni, je parametrem instrukce*/
   int a;    /*cast adresy nebo kod operace, je parametrem instrukce*/
} INSTRUCTION;

/* generovani instrukci do pole code = řetězec postfixových instrukci
      x    :kod instrukce  tj. postfixovy operátor
      y    :hladina
      z    :adresni cast
*/
void gen(FCT x, int y, int z) {

   if (cx>CXMAX) {   /* cx je globální proměnná, citac instrukci*/
      printf("program too long");
      exit(1);
   }
   code[cx].f = x;          /*druh=funkce instrukce*/
   code[cx].l = y;          /*hladina = level*/
   code[cx++].a = z;        /*adresa*/
}   // gen()

void factor(SYMSET fsys,int tx,int lev) {/*násl. symb., konec tab. symb., hladina*/
int i;
SYMSET pom;

   test(facbegsys,fsys,24); /*poč. symboly, násl. symboly, číslo chyby*/
   while (facbegsys[sym]) {
      if (sym == ident) {
         i = position(id,tx); /*najdi ho v tabulce symbolů */
         if (i == 0) error(1);
         else
            switch (TABLE[i].kind) {
                     case constant: gen(lit,0,TABLE[i].CO.val);
                          break;
                             /*identifikátor jako faktor, může být konstanta nebo proměnná*/
                     case variable: gen(lod,lev-TABLE[i].CO.vp.level,TABLE[i].CO.vp.adr);
                          break; /*lev je úroveň místa použití, dále je úroveň deklarace a ofset */
                             /*ale nemůže být jménem podprogramu*/
                     case procedure: error(21);
                           break;
                  }
         getsym();
      } else
         if (sym == number) { /*faktor tské může být pojmenovaným číslem*/
            if (num > AMAX) {
               error(31);
               num = 0;
            }
            gen(lit,0,num);
```

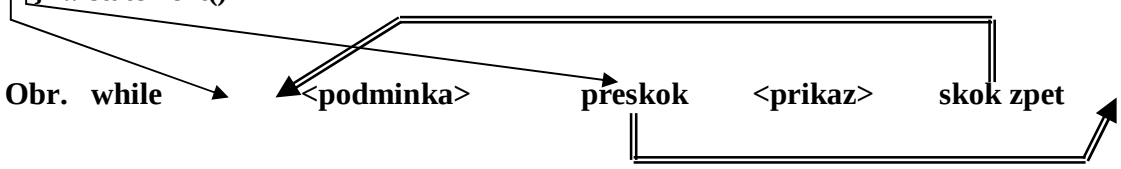
```
            getsym();
         } else
            if (sym == lparen) { /*faktor může být také závorkovaný výraz*/
               getsym();
               nuluj(pom);
               sjednot(pom,fsys);
               pom[rparen] = 1;  /*do follow přidáme ) */
               expression(pom,tx,lev);
               if (sym == rparen) getsym();
               else error(22);
            }
            nuluj(pom);
            pom[lparen] = 1;
            test(fsys,pom,23); /*následující symb., stop symb., číslo chyby*/
   }
}   // factor()

/* následuje generovani kodu pro term, parametry znamenaji
      fsys           :mnozina follow symbolu
      tx   :ukazatel na konec tabulky symbolu
      lev :hladina místa termu
*/
void term(SYMSET fsys,int tx,int lev) {
SYMBOL mulop;
SYMSET pom;

   nuluj(pom);
   sjednot(pom,fsys);
   pom[times] = pom[slash] = pom[modulo] = 1;
   factor(pom,tx,lev);

   while ((sym == times) || (sym == slash) || (sym == modulo)) {
      mulop = sym;
      getsym();
      sjednot(pom,fsys);
      pom[times] = pom[slash] = pom[modulo] = 1;
      factor(pom,tx,lev);

      switch(mulop) {
         case times: gen(opr,0,mul); /*viz Operation  na 1.str programu: mul je 4, di 5,  mod 6 */
                  break;

         case slash: gen(opr,0,di);
                  break;

         case modulo: gen(opr,0,mod);
                   break;
      }
   }

}   // term()
```

```
/* generovani kodu pro aritm. vyraz
      fsys          :mnozina follow symbolu vyrazu
      tx   :ukazatel na konec tabulky symbolu
      lev :hladina mista vyrazu
*/
void expression(SYMSET fsys,int tx,int lev) {
SYMBOL addop;
SYMSET pom;

   if ((sym == plus) || (sym == minus)) {
      addop = sym;
      getsym();
      nuluj(pom);
      sjednot(pom,fsys);
      pom[plus] = pom[minus] = 1;
      term(pom,tx,lev);
      if (addop == minus) gen(opr,0,neg);  /* neg ma v Operation pořadí 1 */
   }
   else {
      nuluj(pom);
      sjednot(pom,fsys);
      pom[plus] = pom[minus] = 1;
      term(pom,tx,lev);
   }

   while ((sym == plus) || (sym == minus)) {
      addop = sym;
      getsym();
      nuluj(pom);
      sjednot(pom,fsys);
      pom[plus] = pom[minus] = 1;
      term(pom,tx,lev);

      if (addop == plus) gen(opr,0,add);  /* pořadí add 2, sub 3  */
      else gen(opr,0,sub);
   }
}  // expression()


/* generovani kodu pro logicky vyraz
      fsys          :follow symboly  vyrazu
      tx   :ukazatel na konec tabulky symbolu
      lev :hladina mista vyrazu
*/
void condition(SYMSET fsys,int tx,int lev) {
SYMBOL relop;
SYMSET pom;

   if (sym == oddsym) {
      getsym();
      expression(fsys,tx,lev);
```

```
      gen(opr,0,odd);
   }
   else {
      nuluj(pom);
      sjednot(pom,fsys);
      pom[eql] = pom[neq] = pom[lss] = pom[gtr] = pom[leq] = pom[geq] = 1;
      expression(pom,tx,lev);
      if ((sym != eql) && (sym != neq) && (sym != lss) && (sym != gtr) && (sym != leq) && (sym !=
geq))
         error(22);
      else {
         relop = sym;
         getsym();
         expression(fsys,tx,lev);

         switch (relop) {
            case eql: gen(opr,0,eq);        // 8
                  break;

            case neq: gen(opr,0,ne);        // 9
                  break;

            case lss: gen(opr,0,lt);        // 10
                  break;

            case geq: gen(opr,0,ge);        // 11
                  break;

            case gtr: gen(opr,0,gt);        // 12
                  break;

            case leq: gen(opr,0,le);        // 13
                  break;

         }
      }
   }

}  // condition()


/* generovani kodu pro statement
      fsys          :follow symboly statementu
      tx   :ukazatel na konec tabulky symbolu
      lev :uroven mista statementu
*/
void statement(SYMSET fsys,int tx,int lev) {
int i, cx1, cx2;
SYMSET pom;

   if ((fsys[sym] == 0) && (sym != ident)) {
```

```
      error(10);
      do
         getsym();
      while (fsys[sym] == 0);
   }
   if (sym == ident) { /*nalezen prikaz prirazeni*/
      i = position(id,tx);
      if (i == 0) error(11);
      else
         if (TABLE[i].kind!=variable) { /*prirazeni do jineho ident. nez promenna*/
            error(12);
            i = 0;
         }
      getsym();
      if (sym == becomes) getsym();
      else error(13);
      expression(fsys,tx,lev);
      if (i) gen(sto,lev-TABLE[i].CO.vp.level,TABLE[i].CO.vp.adr);
   }      /*hladina místa použití minus hladina deklarace, offset*/
   else
      if (sym == callsym) {/*nalezeno volani podprogramu*/
         getsym();
         if (sym != ident) error(14);
                else {
                   if ((i = position(id,tx)) == 0) error(11);
                   else {
                      if (TABLE[i].kind == procedure) gen(cal,lev-
TABLE[i].CO.vp.level,TABLE[i].CO.vp.adr); /*rozdíl úrovní, adresa začátku kódu*/
                                                 else error(15);
            }
                   getsym();
                }
      }
      else
         if (sym == ifsym) { /*podmineny prikaz*/
            getsym();
            nuluj(pom);
            sjednot(pom,fsys);
            pom[thensym] = pom[dosym] = 1;
            condition(pom,tx,lev);

            if (sym == thensym) getsym();
            else error(16);
            cx1 = cx;              /*zapamatování čítače*/
            gen(jpc,0,0);          /*generování neúplné instrukce*/
            statement(fsys,tx,lev);
            code[cx1].a = cx;      /*doplnění adresy*/
         }
         else
            if (sym == beginsym) { /*zacina novy blok*/
               getsym();
```

```
            nuluj(pom);
            sjednot(pom,fsys);
            pom[semicolon] = pom[endsym] = 1;
            statement(pom,tx,lev);

            while ((statbegsys[sym]) || (sym == semicolon)) {
               if (sym == semicolon) getsym();
               else error(10);
               sjednot(pom,fsys);
               pom[semicolon] = pom[endsym] = 1;
               statement(pom,tx,lev);
            }

            if (sym == endsym) getsym(); /*konci predchozi blok*/
            else error(17);
         }
         else
            if (sym == whilesym) { /*zacina cyklus while*/
               cx1 = cx;          /*zapamatujeme čítač instrukcí před podmínkou*/
               getsym();
               nuluj(pom);
               sjednot(pom,fsys);
               pom[dosym] = 1;
               condition(pom,tx,lev);
               cx2 = cx;          /*zapamatujeme číslo instrukce skoku ven z cyklu*/
               gen(jpc,0,0);      /*generujeme instrukci skoku, zatím nevíme kam*/
               if (sym == dosym) getsym();
               else error(18);
               statement(fsys,tx,lev);
               gen(jmp,0,cx1); /*generujeme skok na testování podmínky*/
               code[cx2].a = cx; /*doplníme teď známou adresu místa za cyklem*/
            }

      test(fsys,pom,19); /*test koreknost ukonceni statementu*/

   }  // statement()
```



Obr.

```
var i,j;
procedure p;
          begin i := i-1; if i>1 then call p;
          end;
begin
         i:=3;
         call p;
end.
```

generovany kod:
****

```
 0 JMP    0   13      skok na začátek hl progr, přeskok deklarací
1 JMP    0    2
2 INT    0    3      začátek procedury p. Rezervace 3 míst, viz kap.5
3 LOD    1    3      natažení i (1 je rozdil mista pouziti a deklarace)
4 LIT    0    1      natažení konstanty 1
5 OPR    0    3      odečtení
6 STO    1    3      uložení i
7 LOD    1    3      natažení i
8 LIT    0    1      natažení 1
9 OPR    0   12      větší
10 JMC    0   12     některé verze používají místo JPC JMC
11 CAL    1    2     1 je rozdíl místa použití a místa deklarace, 2 začátek
12 RET    0    0     návrat, nekteré verze překladače používají OPR 0  0
13 INT    0    5     rezervace míst pro stat a dyn ukaz, návr adr, i, j
14 LIT    0    3     natažení konstanty 3
15 STO    0    3     uložení trojky do i
16 CAL    0    2     vyvolání procedury p
17 RET    0    0
```

tabulka symbolu:
****

```
  1  name: i        var   lev=    0   adr=    3  size=    0
  2  name: j        var   lev=    0   adr=    4  size=    0
  3  name: p        proc  lev=    0   adr=    2  size=    3
```

Složitější příklad s vnořeným podprogramem:

```
var a,aa;procedure p1;
          var b;
          procedure p2;
          begin a := 10; b:=20;
        end;
      begin call p2; a:=b*b;
      end;
begin call p1;aa:=a
end.
```

generovany kod:

****

```
  0 JMP     0    16
  1 JMP     0     9
  2 JMP     0     3
  3 INT     0     3
  4 LIT     0    10
  5 STO     2     3
  6 LIT     0    20
  7 STO     1     3
  8 RET     0     0
  9 INT     0     4
 10 CAL     0     3
 11 LOD     0     3
 12 LOD     0     3
 13 OPR     0     4
 14 STO     1     3
 15 RET     0     0
 16 INT     0     5
 17 CAL     0     9
 18 LOD     0     3
 19 STO     0     4
 20 RET     0     0
```

tabulka symbolu:

****

```
  1  name: a          var   lev=    0   adr=    3  size=    0
  2  name: aa         var   lev=    0   adr=    4  size=    0
  3  name: p1         proc  lev=    0   adr=    9  size=    4
  4  name: b          var   lev=    1   adr=    3  size=    0
  5  name: p2         proc  lev=    1   adr=    3  size=    3
```

Zpracování deklarací a přidělování paměti

- **Účel deklarací**
  -pojmenování objektů
  -umístění objektů v paměti

- **Tabulka symbolů**
  -uchovává informace o objektech
  -umožňuje kontextové kontroly
  -umožňuje operace
    1. **inicializaci informace pro standardní jména**
    2. **vyhledání jména**
    3. **doplnění informace ke jménu**
    4. **přidání položky pro nové jméno**
    5. **vypuštění položky či skupiny položek**

Struktura tabulky symbolů

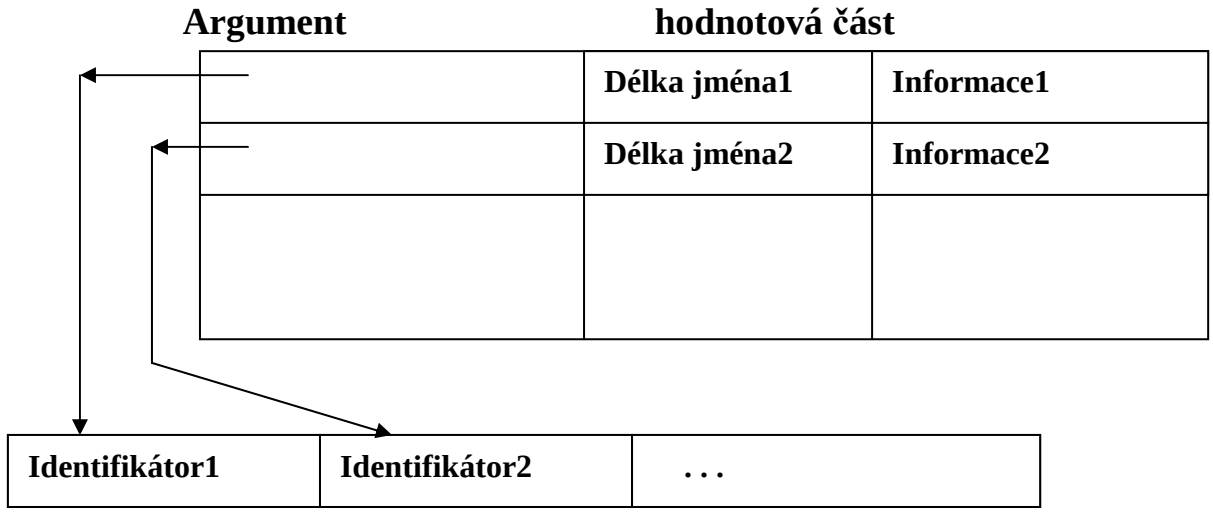
- **s jednoduchou strukturou**
- **s oddělenou tabulkou identifikátorů**
- **s oddělenou tabulkou informací**
- **uspořádané do podoby zásobníku**
- **s blokovou strukturou**

Tabulka symbolů:

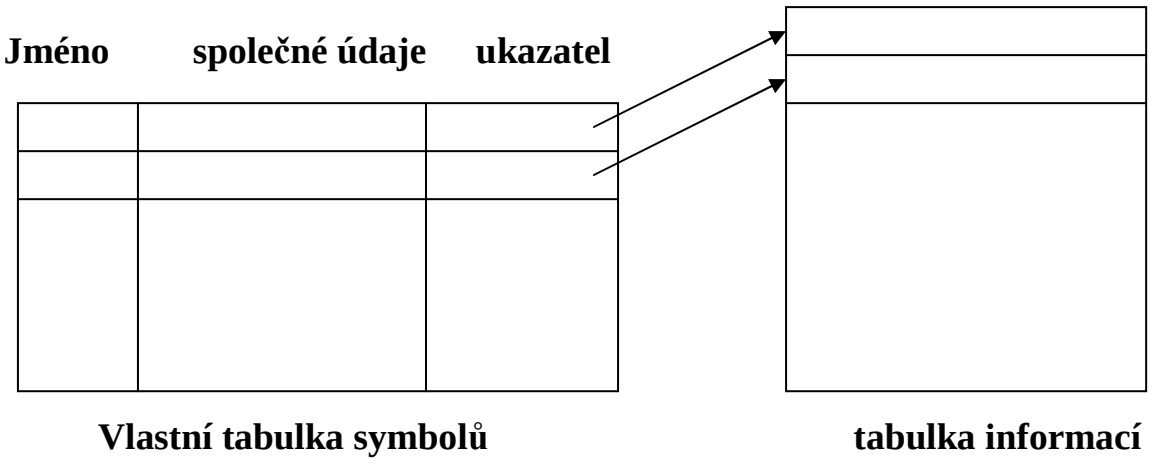
Argument= jméno | hodnotová část= atributy

1.položka
2.položka
.
.
.
n-tá položka

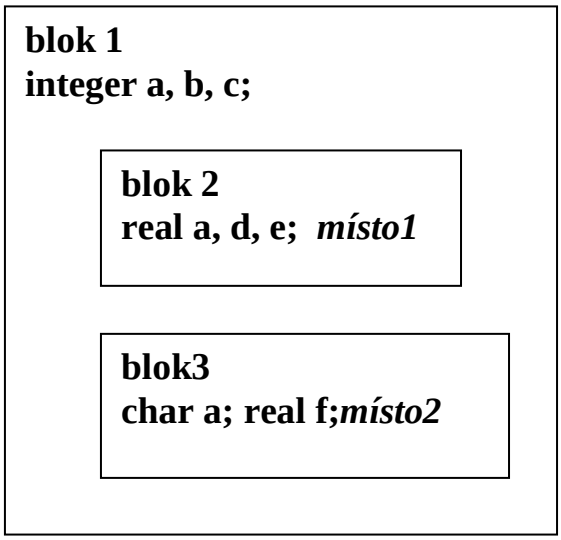
Tabulka symbolů s oddělenou tabulkou identifikátorů (neomezená jména)

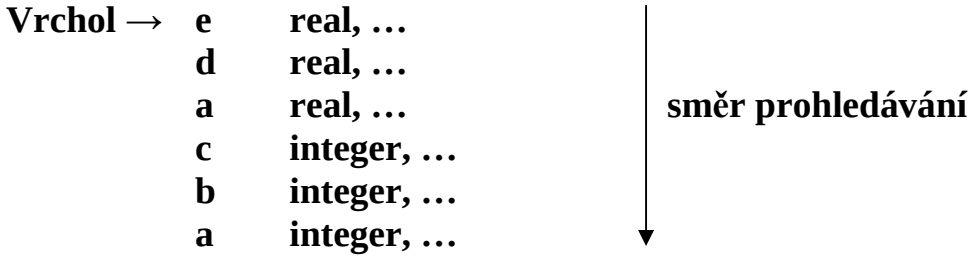


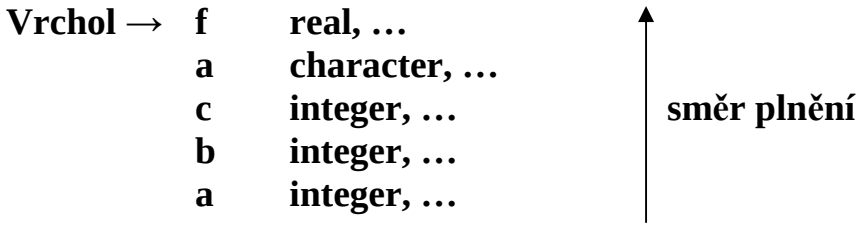
Tabulka symbolů s oddělenou tabulkou informací



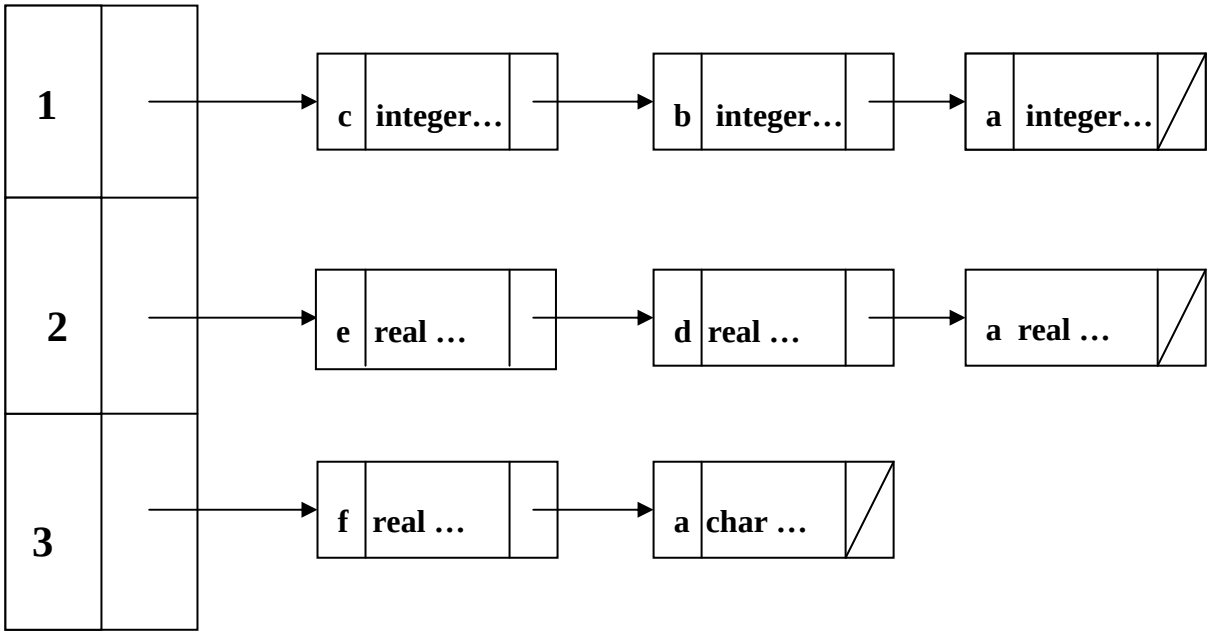
Tabulka symbolů uspořádaná do podoby zásobníku (pro jazyky s blokovou strukturou).
Rozsahová jednotka je blok, modul, funkce, balík,…
Respektuje zásady lokality

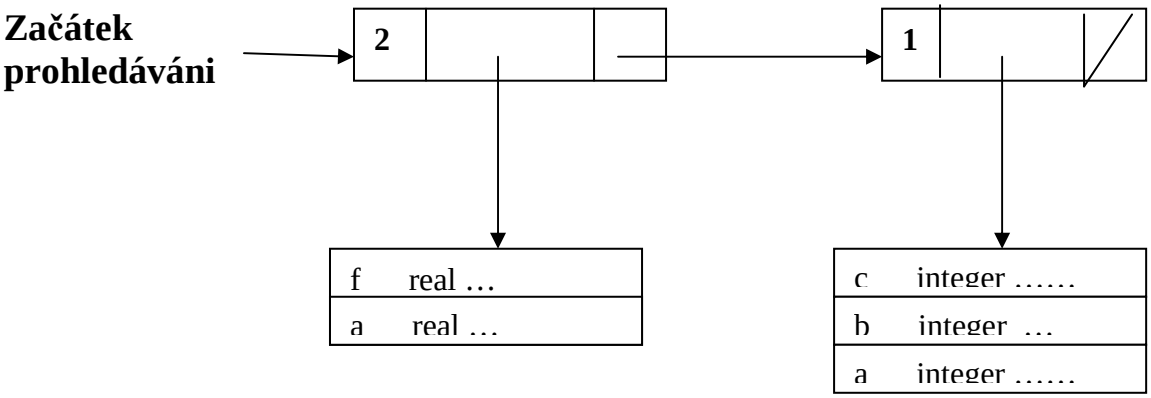






Tabulka symbolů s blokovou strukturou





Diskutujte
? jak řešit případ, kdy jazyk dovoluje použít jména před jejich deklarací
? jak řešit případ, kdy jazyk dovoluje přetěžování jmen

Informace v tabulce symbolů
(závisí na jazyce i způsobu překladu)

Př. pro jazyk s blokovou strukturou

1. **DRUH**
   - **-návěští**
   - **-konstanta**
   - **-typ i příp. objektový**
   - **-proměnná**
   - **-procedura**
   - **-funkce**
   - **-metoda**

2. **Hladina popisu = úroveň vnoření**

3. **Adresa**
   - **-funkcí, procedur, metod**
   - **-proměnných (statická nebo offset)**
   - **-konstant**

4. **Použití byla/nebyla použita**

5. **Typ**
   - **-údaj o standardním jednoduchém typu**
   - **-údaj o typu definovaném uživatelem**
   - **-údaj o strukturovaném typu**
   - **-údaj o objektovém typu**

6. **Formální parametr je/není to formální parametr**

7. **Druhy formálních parametrů kolik jich má a kde jsou v TS**

8. **Způsob volání hodnotou/odkazem**

9. **Hodnota jen u konstant**

Implementace tabulky symbolů

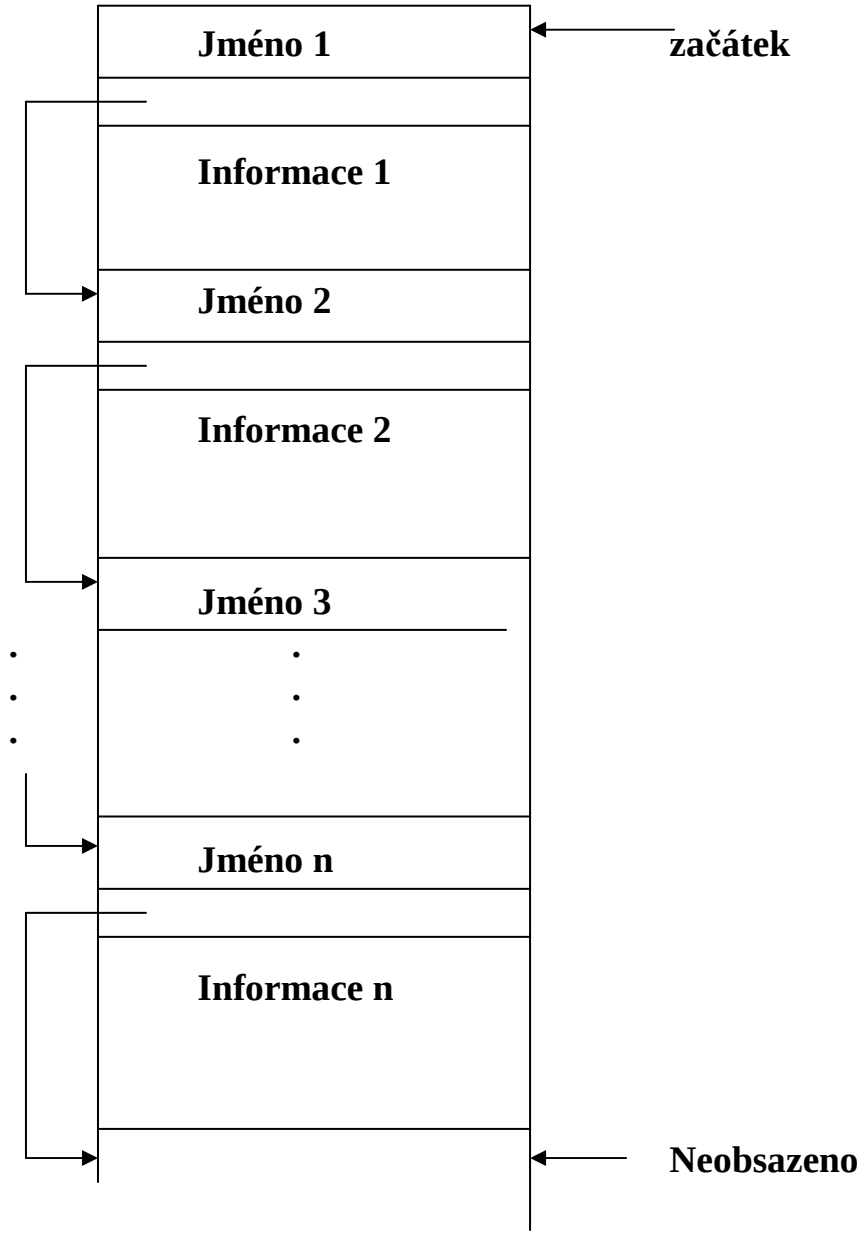
- **Vyhledávací netříděné tabulky (jen pro krátké programy)**
  -prostá struktura
  -lineární seznam

- **Vyhledávací setříděné tabulky**
  -průběžné setřiďování
  -setřídění po zaplnění

- **Frekvenčně uspořádané tabulky**

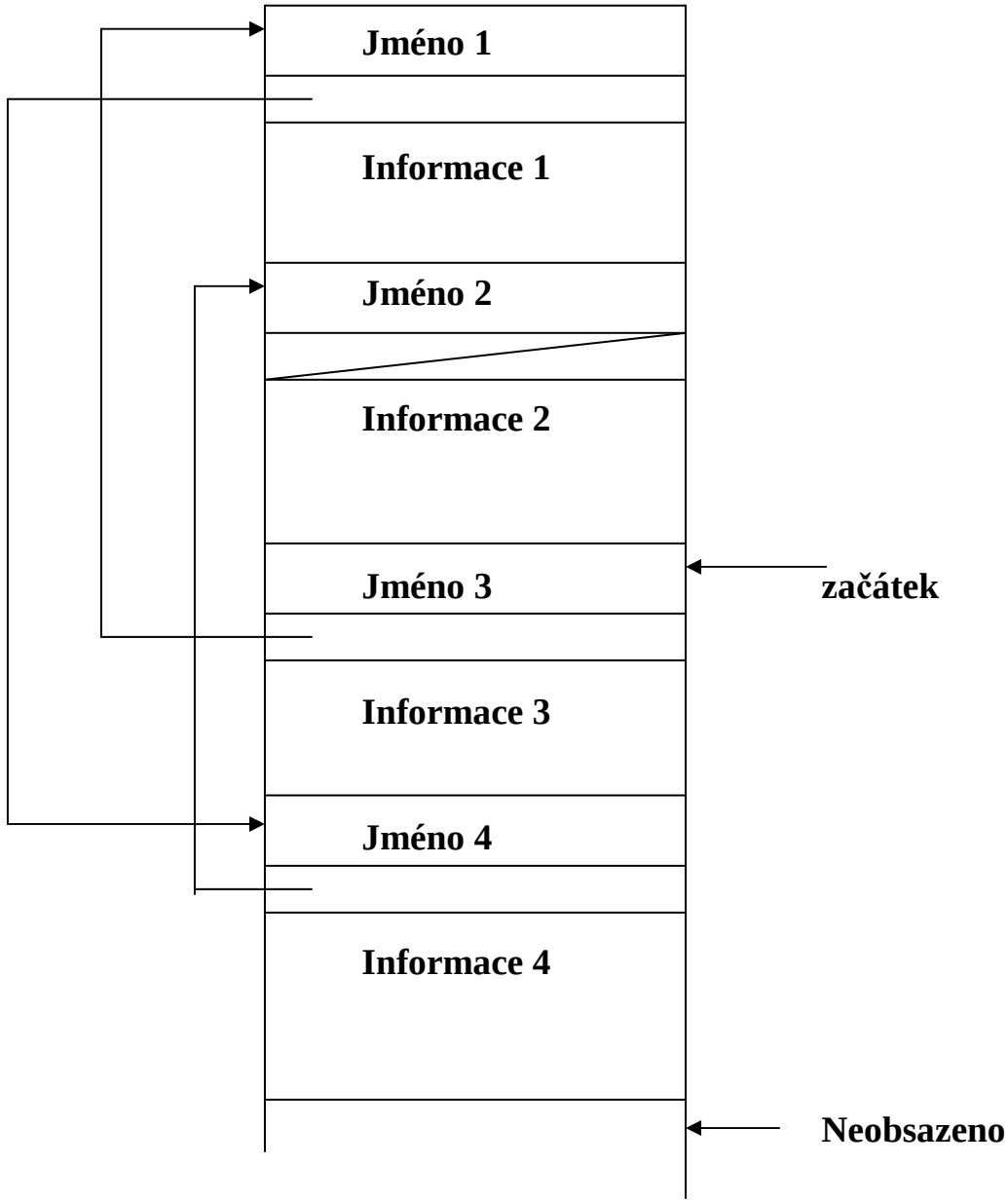
- **Binární vyhledávací stromy**

- **Tabulky s rozptýlenými položkami**



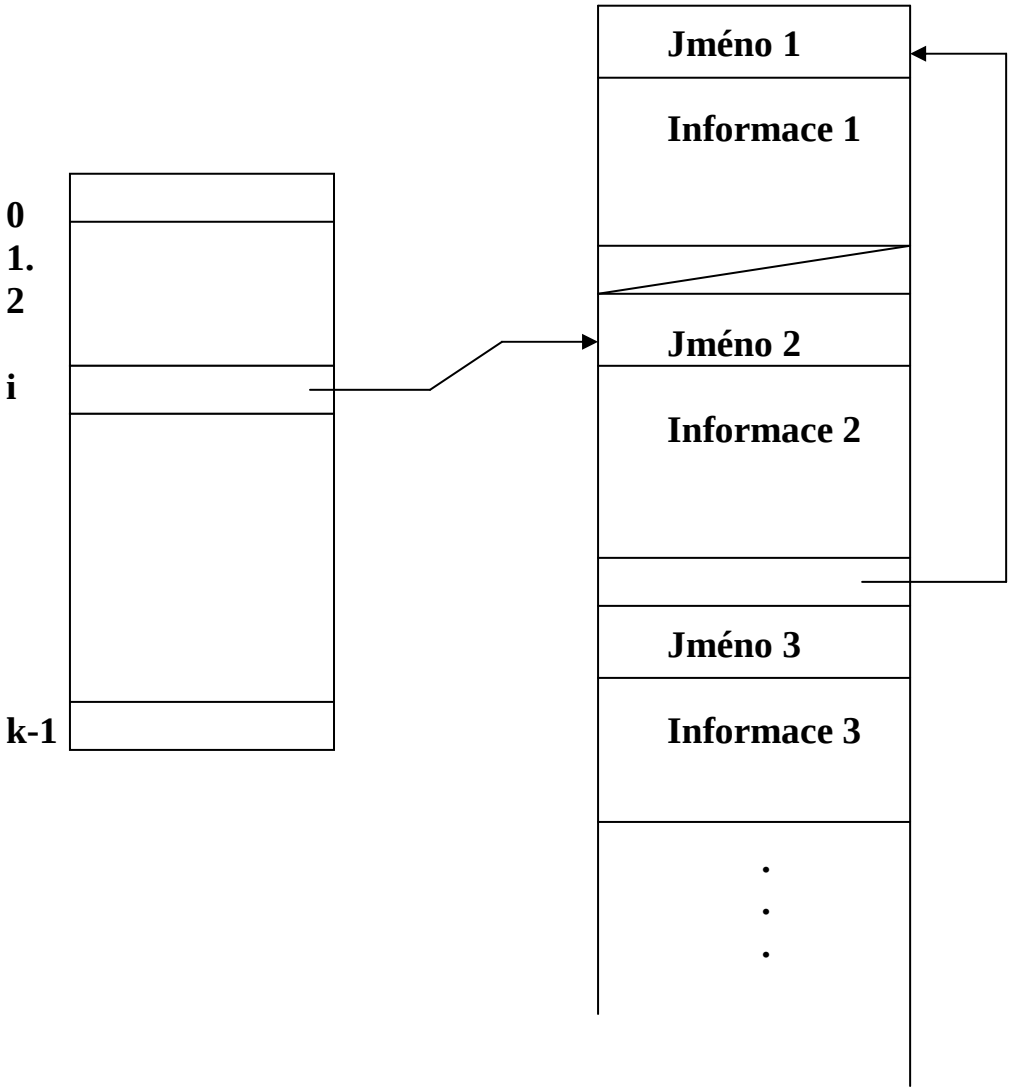
Tabulka symbolů organizovaná jako lineární seznam

? Jaká je časová náročnost v závislosti na počtu položek?



Frekvenčně uspořádaná tabulka symbolů

? Jaká je časová náročnost v závislosti na počtu položek?



Rozptýlená organizace tabulky symbolů

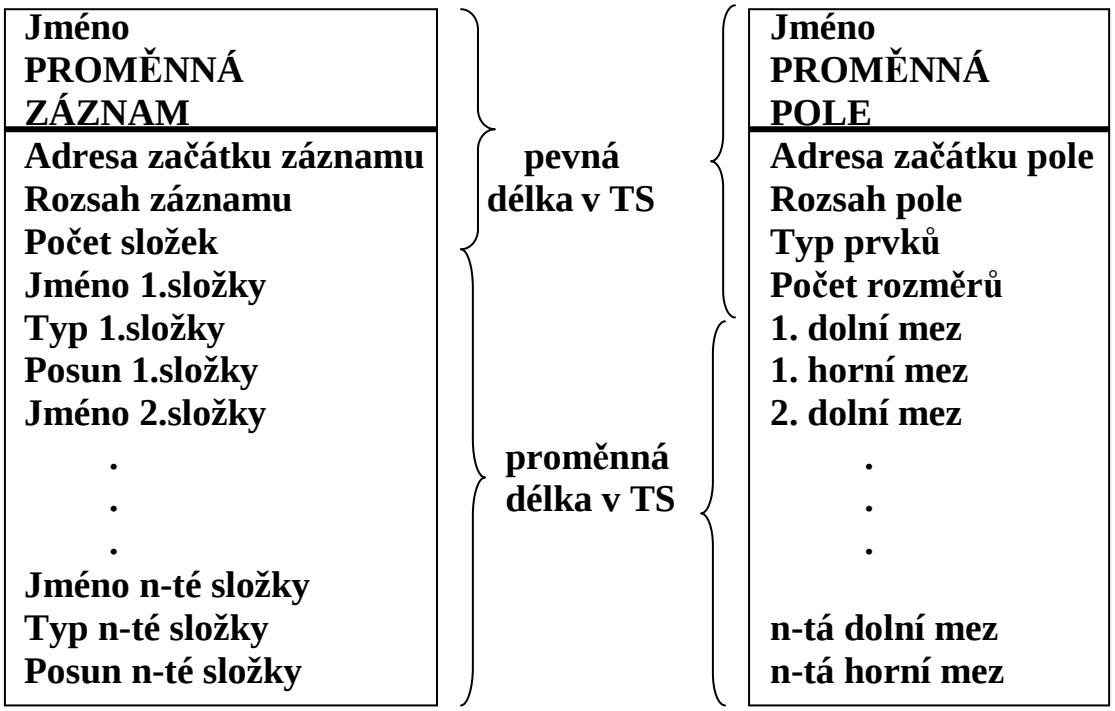
- **-Jméno upravíme na n znaků**
- **-Rozptylovací funkce h(jméno)=zbytek po dělení (∑ord(znak))/k**
- **-Řešení kolizí**
- **-Výpočtová složitost**

? Jaká je časová náročnost v závislosti na počtu položek?

Překlad deklarace datových struktur

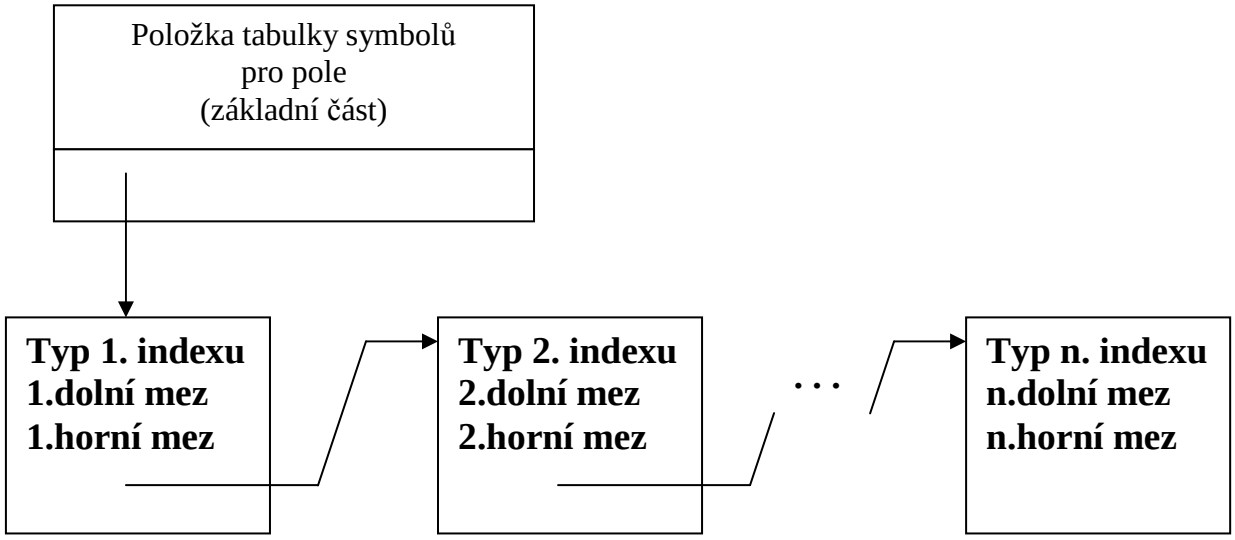
Údaj o uspořádání strukturovaného typu se nazývá deskriptor

Statické struktury – deskriptor lze celý vytvořit a dát do TS při překladu
Dynamické struktury – hodnoty v deskriptoru se plní v čase výpočtu



A) Položka TS a deskriptor záznamu **B) Položka TS a deskriptor pole**

Alternativní struktura položky TS pro pole
(údaje o dimenzích jsou řetězeny do seznamu)



Ukládání polí

Předp. deklaraci tvaru: **array[D1..H1, D2..H2, …, Dn..Hn] of T**

Adresa prvku A[i1, i2, …in] **=** **adresa začátku pole**
+ hodnota mapovací fce

Adresa začátku pole = adresa prvku s nejnižšími indexy
Mapovací fce = posun prvku vzhledem k začátku pole

$$\text{hod.map.fce.prvku [i1,i2,…in]} = \sum_{k=1}^{n} (ik - Dk) * Kk$$

konstanty=koef.indexů
dolní meze indexů

pro pm(T) = počet míst paměti k uložení prvku typu T platí:

a) při uložení pole po řádcích

$K_n = pm(T)$

$K_k = (H_{k+1} - D_{k+1} + 1) * K_{k+1}$ **pro k= n–1, …, 2, 1**

b) při uložení pole po sloupcích

$K_1 = pm(T)$

$K_k = (H_{k-1} - D_{k-1} + 1) * K_{k-1}$ **pro k=2, 3, …, n**

Mapovací funkci lze rozdělit

$$\text{Mapovací funkce} = \sum_{k=1}^{n} (ik*Kk) - \sum_{k=1}^{n} (Dk*Kk)$$

konstantní částí je $(\text{adresa\_začátku pole} - \sum_{k=1}^{n} (Dk*Kk))$

?co je to za adresu?

Deskriptor pole zahrnuje:
1)typ prvků, 2)počet rozměrů, 3)meze indexů, 4)koeficienty map.fce, 5)konstantní část map.fce, 6)velikost paměti pro pole
U dynamického pole se 3-6 dopočítají při běhu programu
U statických je součástí tabulky symbolů

Př.

Pole: **a21 a22 a23 a24 a25**
a31 a32 a33 a34 a35

nechť je: Pm(T) = 10 počet paměť.míst pro typ T
Začátek pole nechť je na 100

Uložení po řádcích: 100-109 110-119 … 140-149
150-159 160-169 … 190-199

D1=2, H1 =3
D2=1, H2 =5
K2=Pm(T)=10 tj. sloupce jsou posunuté o 10
K1=(H2-D2+1)xK2=(5-1+1)x10 = 50 tj. řádky posunuté o 50
Konstantní část = adr.začátku - (D1xK1 + D2xK2) =
100 -110 = -10

Např. a22 bude na adrese i1 x K1 + i2 x K2 + konst.část
=
=2 x 50 + 2 x 10 – 10 = <u>110</u>

Uložení po sloupních: 100-109 120-129 … 180-189
110-119 130-139 … 190-199

K1 = 10
K2 = (H1-D1 + 1) x K1 = (1+1) x 10 = 20
Konst. část = 100 – (2 x 10 + 1 x 20) = 60

Např a22 bude na adrese 2 x 10 + 2 x 20 + 60 = <u>120</u>

Ukládání záznamů

Předp. deklaraci tvaru: struct p1:T1, p2:T2, …, pn:Tn end;

Přístup k položce p záznamu Z

$$adresa(Z.p) = adresa\_začátku(Z) + posun(p)$$

Pro posun položky pi platí:

$$Posun(pi) = \sum_{j=1}^{i-1} rozsah(Tj)$$

Překlad objektových konstrukcí

Deklarace třídy:

```
class JMENOTRIDY extends SUPERCLASSJMENO {
        datovepolozky     /* jako záznamy*/
        metody       /*maji implicitni parametr this typu JMENOTRIDY /*
}  /* pri prekladu se vytvori descriptor tridy v TS */
```

Deskriptor třídy obsahuje

- Ukazatel na deskriptor rodiče
- Seznam datových položek
  - jejich ofset
  - údaj o privat
- Seznam metod
  - jejich vstupní bod
  - údaj o static
  - údaj o privat

Vytvoření objektu **včetně inicializace jeho datových prvků lze provést:**

```
a) objektpromenna = new JMENO TRIDY;
        /* při výpočtu vytvoří na haldě   ClassInstanceRecord (CIR)*/
        /* při překladu      „     v TS          jméno
                                                 typ objekt
                                                 odkaz na deskriptor tridy
        */
b) deklarací:  jméno_třídy  jméno_instance; // alokuje se v zásobníku
        /*při zpracování deklarace výpočtem se vytvoří CIR, jeho zásobníková
                adresa a velikost byla určena při překladu
        */
```

Volání metody

```
objektpromenna . metoda (parametry) ;
```

-U statických hledá se při překladu vstupní bod v deskriptoru třídy, nenajde-li se, pak v deskriptoru nadtřídy, …

-U dynamických obsahuje descriptor třídy ještě ukazatel na tabulku virtuálních metod (VMT). Před prvým voláním metody je nutné vyvolat konstruktor. Ten propojí instanci volající konstruktor s VMT.

Odkaz na datovou položku

```
objektpromenna . datovapolozka    // stejný mechanismus jako záznamy
```

Jednoduchá dědičnost datových položek

Potomek má zděděné položky na začátku svého rekordu, v pořadí jako u rodiče, nedělené položky jsou uloženy za nimi.

Př. **class R { int x = 0;}**
class P1 extends R { int y = 0; int z = 0; int u = 0; }
class P11 extends P1 { int v = 0; }
class P2 extends R {int y = 0; }

R	P1	P2	P11
x	x	x	x
	y	y	y
	z		z
	u		u
			v

V podstatě jako záznam, s rozdílem

- **lze použít objekt typu potomka i tam, kde se očekává předek překladač to zkontroluje podle údajů v tab.symbolů**

Překlad metod - obdoba překladu funkcí

Statické metody

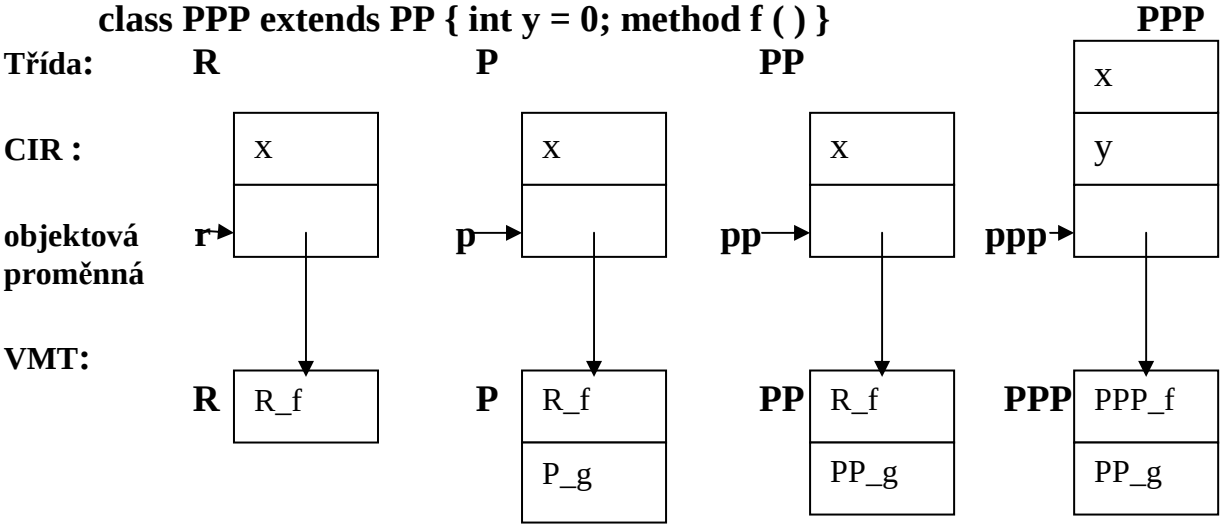
c . f( ) **překlad volání f závisí na typu objektové proměnné c**

-překladač vyhledá třídu C objektu c
-v C hledá metodu f
-když jí nenajde, hledá jí v rodiči C (informaci má v deskriptoru)
-…
-pokud program není chybný, v nějakém předkovi jí najde
-volání se přeloží do skoku na její vstupní bod.

Dynamické metody

Mohou být překryty ve třídě potomka.
V době překladu nelze určit zda se jedná o metodu předka či potomka

Př. **class R { int x = 0; method f ( ) }**
class P extends R { method g ( ) }
class PP extends R { method g ( ) }
class PPP extends PP { int y = 0; method f ( ) }



R_f označuje metodu f z třídy R, …
Ukazatel na VMT se získá pro CIR pomocí deskriptoru třífy.
Je-li proměnná pp ukazatel na objekt z třídy PP (může ukazovat na objekt třídy PP a jejích potomků), nelze určit při překladu zda volání pp . f ( )
-je voláním R_f -ukazuje-li pp na objekt třídy PP
-nebo voláním PPP_f -ukazuje-li pp na objekt třídy PPP

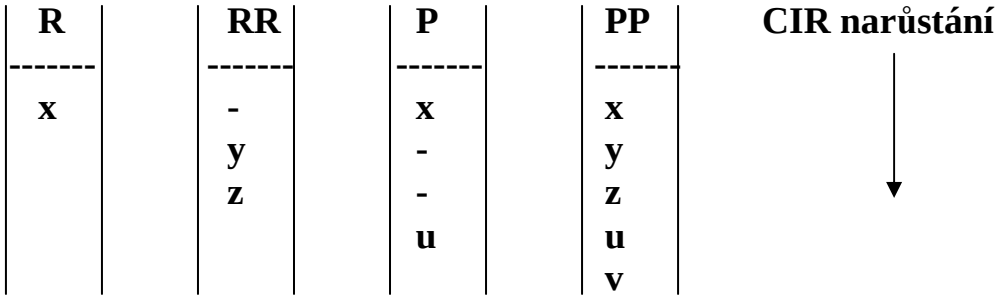
Řešení:

- **Deskriptor třídy musí obsahovat vektor s metodami pro každé ze jmen nestatických metod (VMT)**
- **Když třída P dědí z R, pak VMT pro P začíná se vstupními body všech metod platných pro R a pokračuje s novými metodami zavedenými v P**
- **Pokud třída PPP překryje metodu R_f, bude na místě R_f ve VMT pro PPP adresa PPP_f**
- **Při exekuci pp . f ( ) musí přeložený kód provést**
  1. **Zjistit deskriptor třídy objektu na který ukazuje pp,**
  2. **Z něj zjistit adresu vstupního bodu metody f**
  3. **Skok do podprogramu na adresu tohoto vstupního bodu**

CIR musí ukazovat na deskriptor třídy, ten se uchovává i v run-time

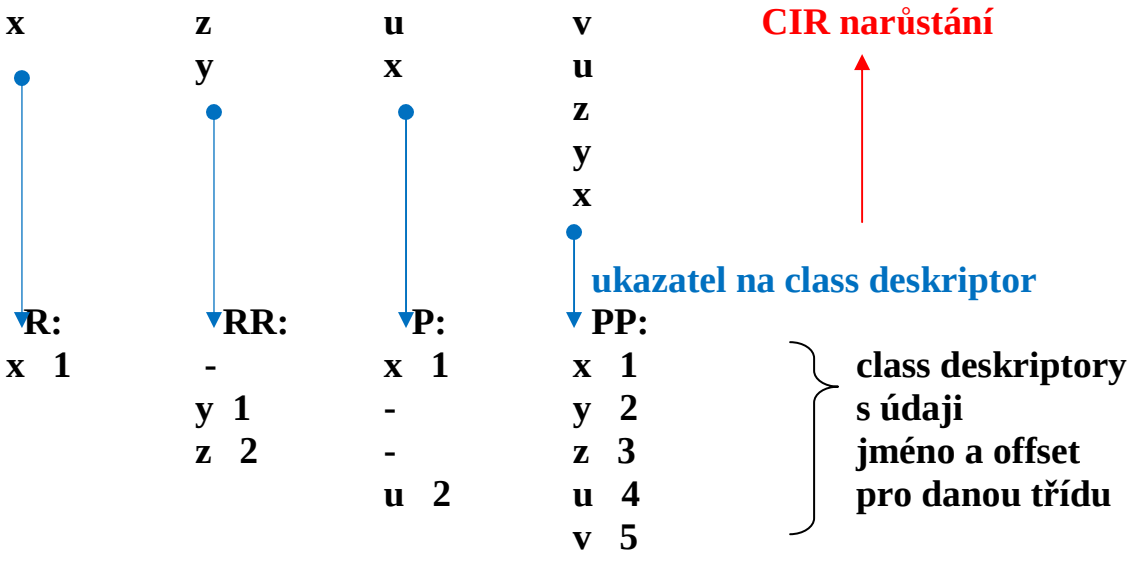
<u>Násobná dědičnost</u> (C++, Smalltalk, Python)
- Python upřednostní dříve uvedené
- Nelze vložit na začátek deskriptoru potomka jak položky rodiče R1, tak i rodiče R2, nutno zabránit opakovanému dědění.
Obtížné nalezení ofsetu položek a metod
Řešitelné např. barvením grafu
- **uzly jsou jména položek**
- **hrany spojují koexistující položky třídy**
- **barvy jsou ofsety (0, 1, 2, …)**

Př. **class R { int x =0 }**
class RR { int y =0; int z = 0 }
class P extends R { int u = 0 }
class PP extends R, RR, P { int v = 0 } //tady existují všechny =5barev



Zůstávají pak prázdné sloty v objektech (v CIR)

Řešení je komplikované:
Zapakovat položky v CIR každého z objektů a zaznamenat ofsety v class deskriptoru.



Ušetří paměť, ale potřebuje více operací:

1. **Vybrat z CIR ukazatel na deskriptor**
2. **Zjistit z deskriptoru ofset**
3. **Použít zjištěný ofset k přístupu k položce**

Deskriptory tříd pak mají i nadále prázdné sloty, ale vlastní objekty ne.

Metody přidělování paměti

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

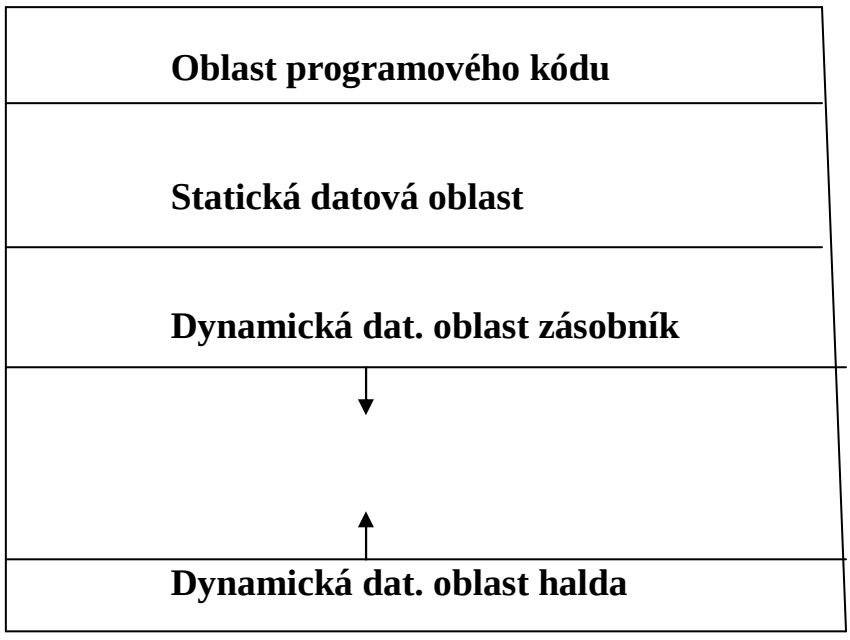
Důležitá hlediska jazykových konstrukcí:

- **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



Statické přidělování paměti lze použít pro:

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

Př.

Blok1
místoA
Blok2
místoB
Blok3
místoC
Konec bloku3

Blok4
místoD
Konec bloku4
místoE
Konec bloku2
místoF
Konec bloku1

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**

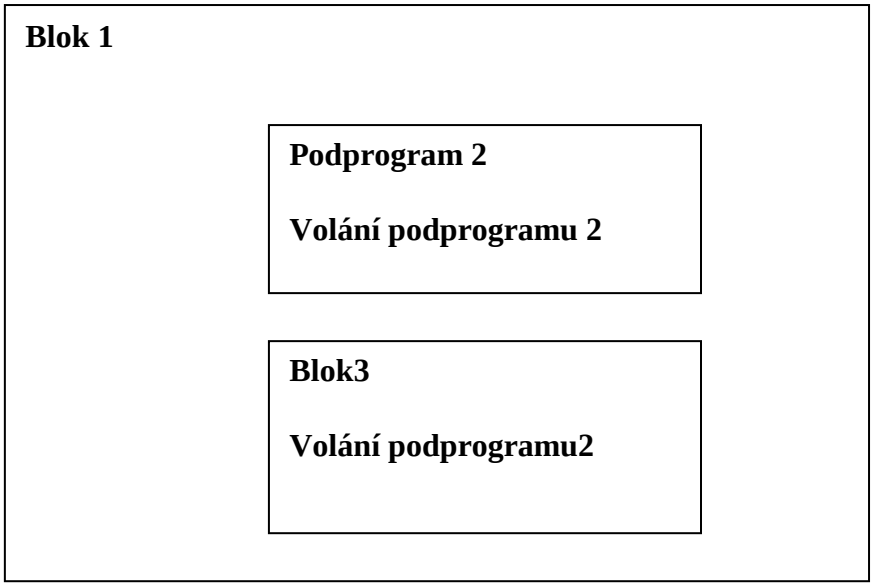
Vnořování podprogramů (funkcí, metod) je ale složitější, viz dále.

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:

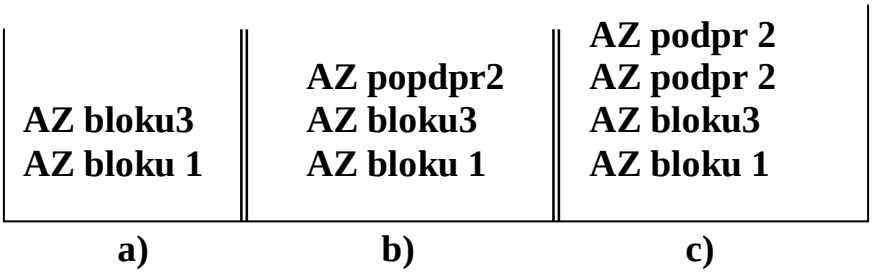
- **Lokální proměnné**
- **Parametry (je-li rozsahovou jednotkou podprogram či funkce)**
- **Návratovou adresu („ „)**
- **Funkční hodnotu (je-li rozsahová jednotka funkcí)**
- **Pomocné proměnné pro mezivýsledky (také možno v registrech)**
- **Další informace potřebné k uspořádání aktivačních záznamů**

Př.1

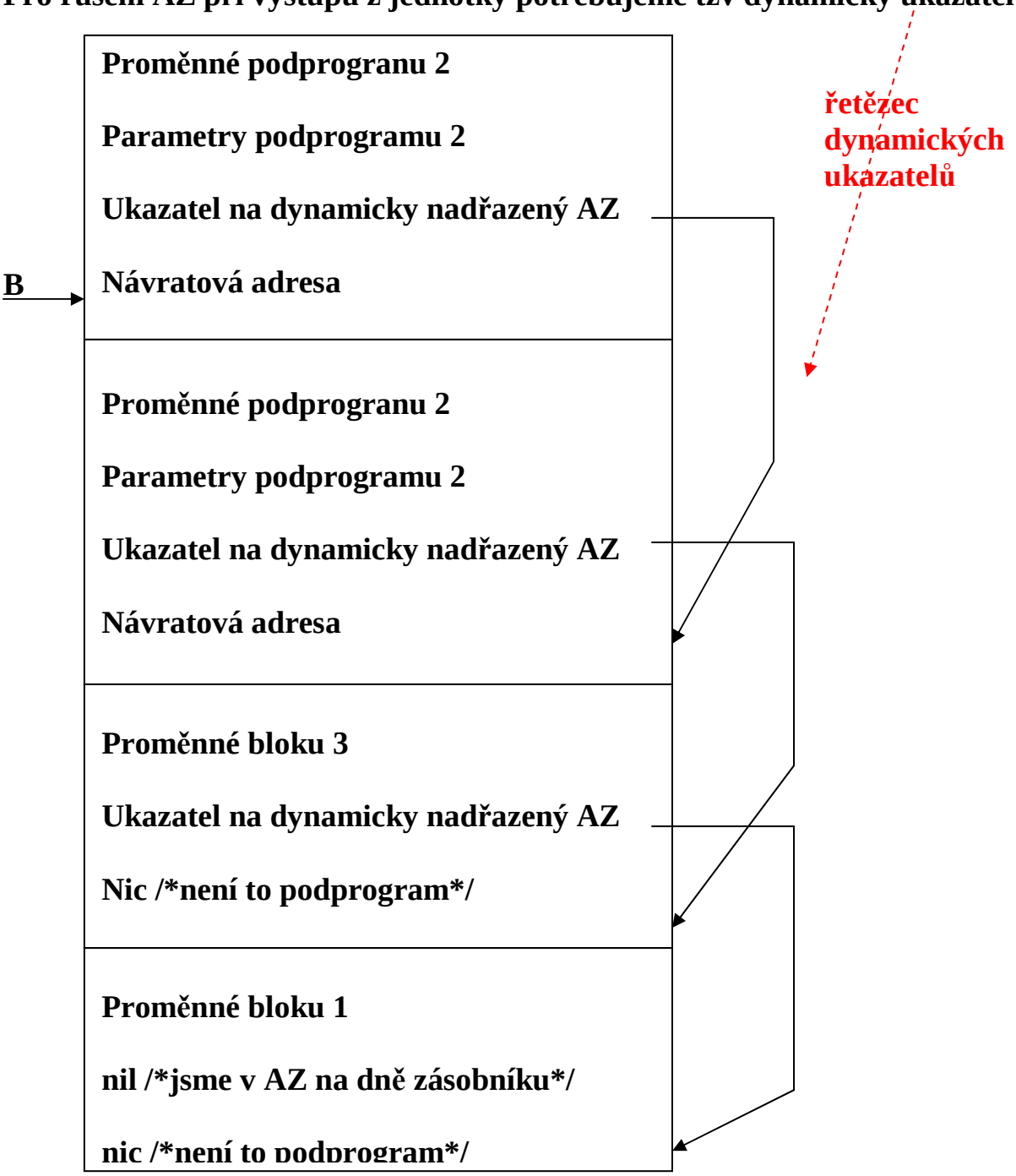


? stav výpočtového zásobníku v různých časech výpočtu

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



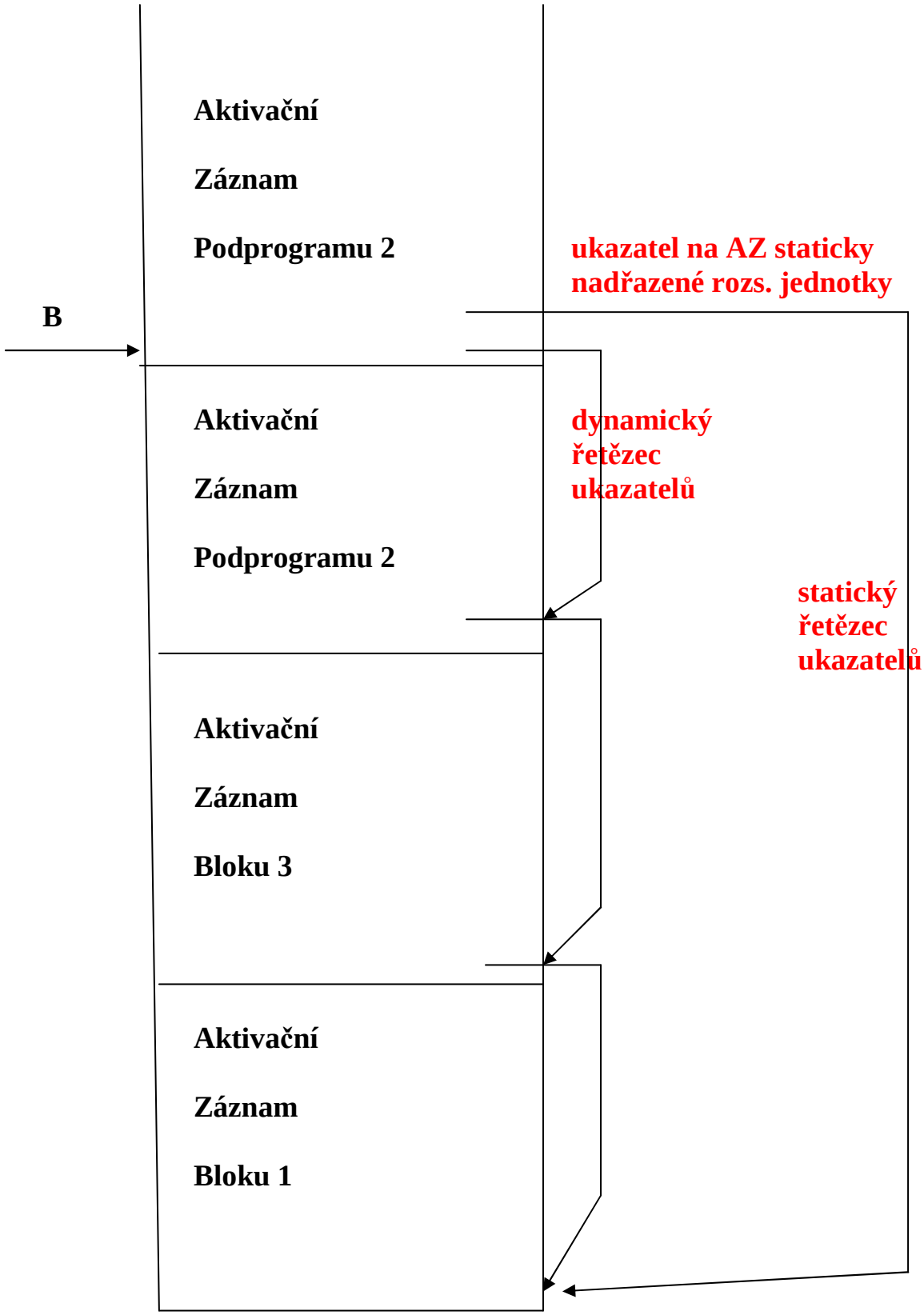
Jaké bude uspořádání aktivačních záznamů při rekurz. vyvolání podpr.2 ?
Pro rušení AZ při výstupu z jednotky potřebujeme tzv dynamický ukazatel



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

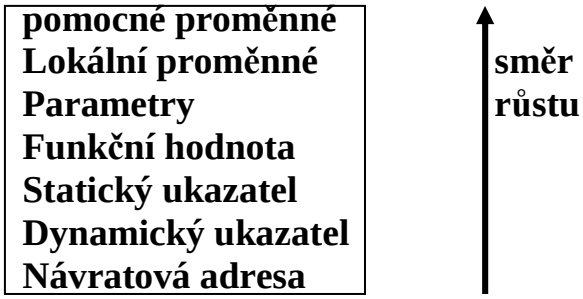


Obr. Zásobník se statickým (ukazuje na lexikálně nadřazený AZ) a dynamickým řetězcem ukazatelů při rekurzivním volání podprogramu 2

Pozn.: Statický uk. je nakreslen (pro přehlednost) jen u AZ rek. volání podpr.2

Vytváření řetězců ukazatelů

Nechť AZ má tvar:



Uvažujme zásobník Z, s vrcholem (nejvyšší zabranou adresou) T.

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

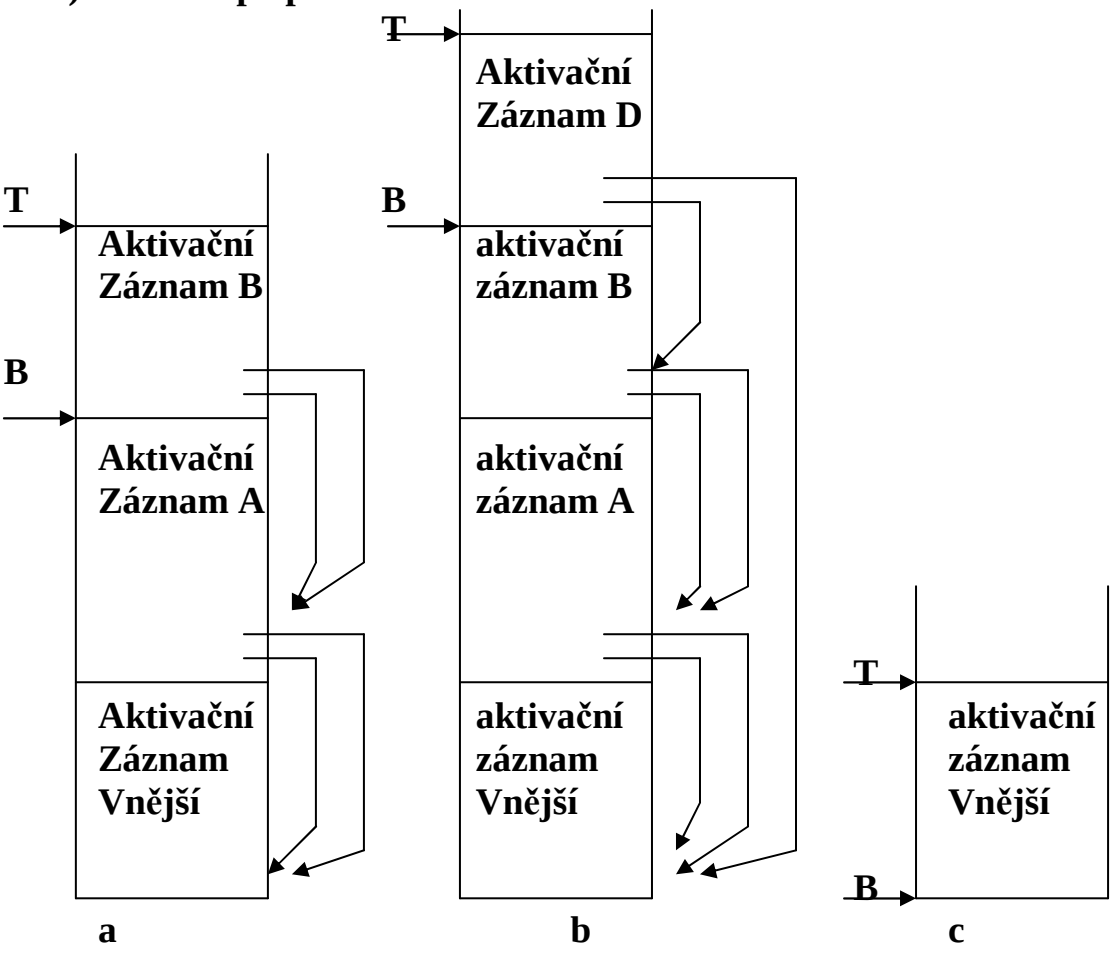
```
S1) for i ← 1 to m – n do { Pom ← B
                            repeat   T ← B – 1
                                     B ← Z[ B + 1]
                            until B ≠ Z[ POM + 2]
                          }
```

S2) skok na adresu, kterou návěští představuje

Př.2

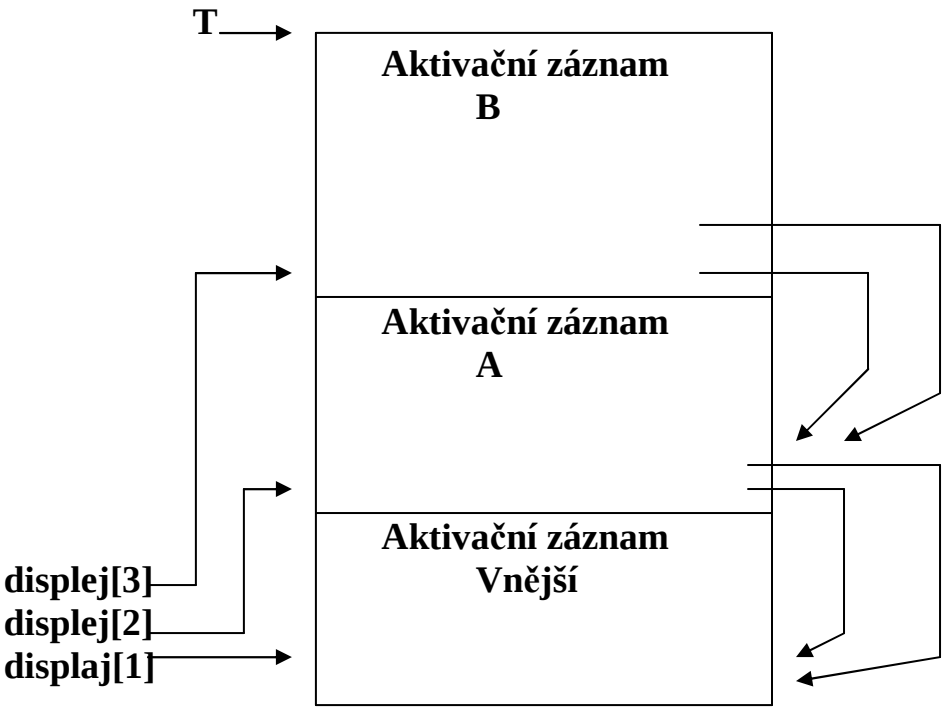
```
Podprogram Vnější //označme n hladinu deklarace
Label 1                    // a m hladinu místa použití
        Podprogram A
                Podprogram B
                Volání D  //zde  n=1; m=3
        Volání B
        Podprogram D
        Goto 1        //zde n=1; m=2
Volání A
1:
```

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.

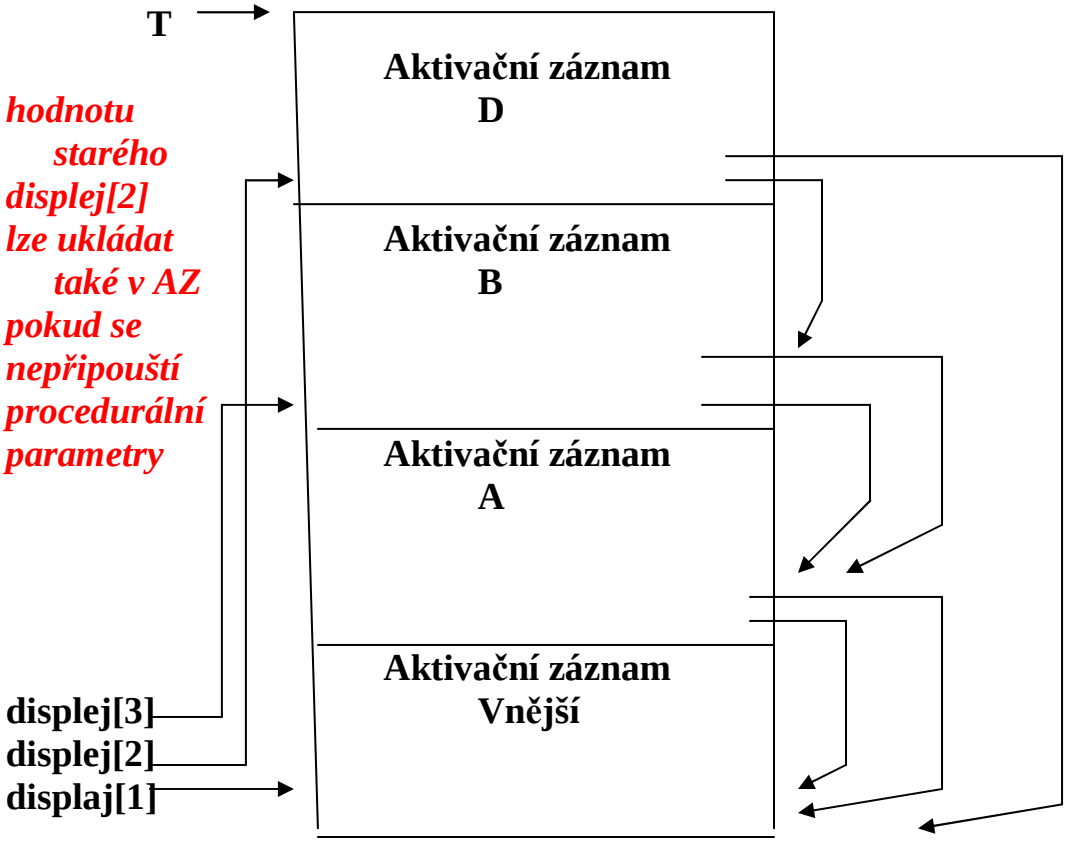


ad b) je to stav v okamžiku volání podpr. s hladinou deklarace 1, volaného v místě s hladinou 3
ad c) stav po výskoku z hladiny 2 do místa s hladinou 1

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



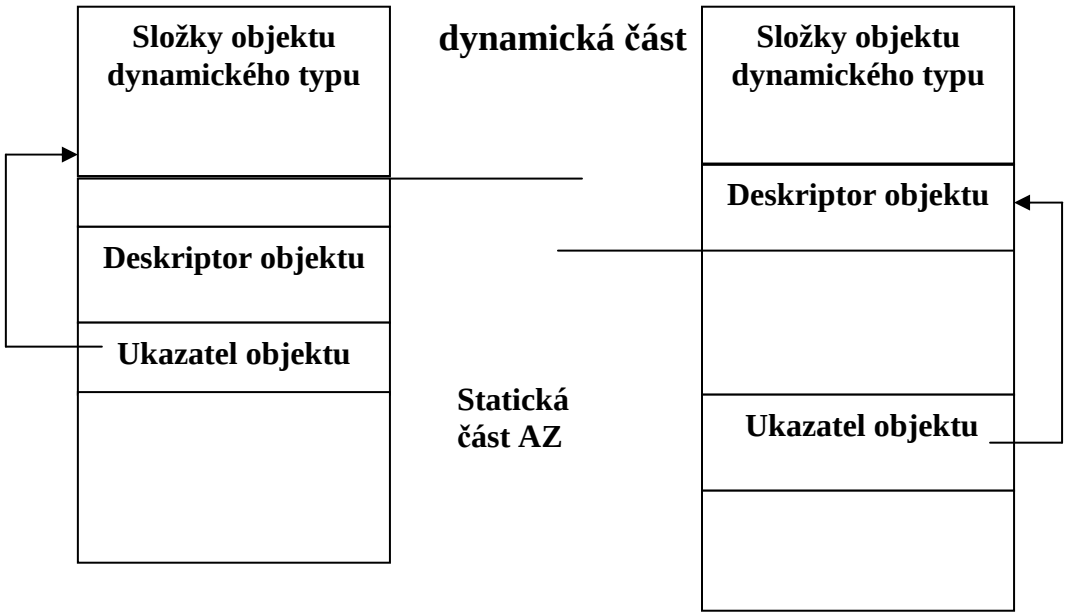
hodnotu starého displej[2] lze ukládat také v AZ pokud se nepřipouští procedurální parametry

Obr. Stav Z při výpočtu D z př.2

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

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,  );
```

dynamická část

místo pro prvky pole B

místo pro prvky pole B

Descriptor B
Ukazatel na prvky B

statická část

Descriptor A
Ukazatel na prvky A

i

j

Parametry podprogramu
Statický ukazatel
Dynamický ukazatel
Návratová adresa

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

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 … ?

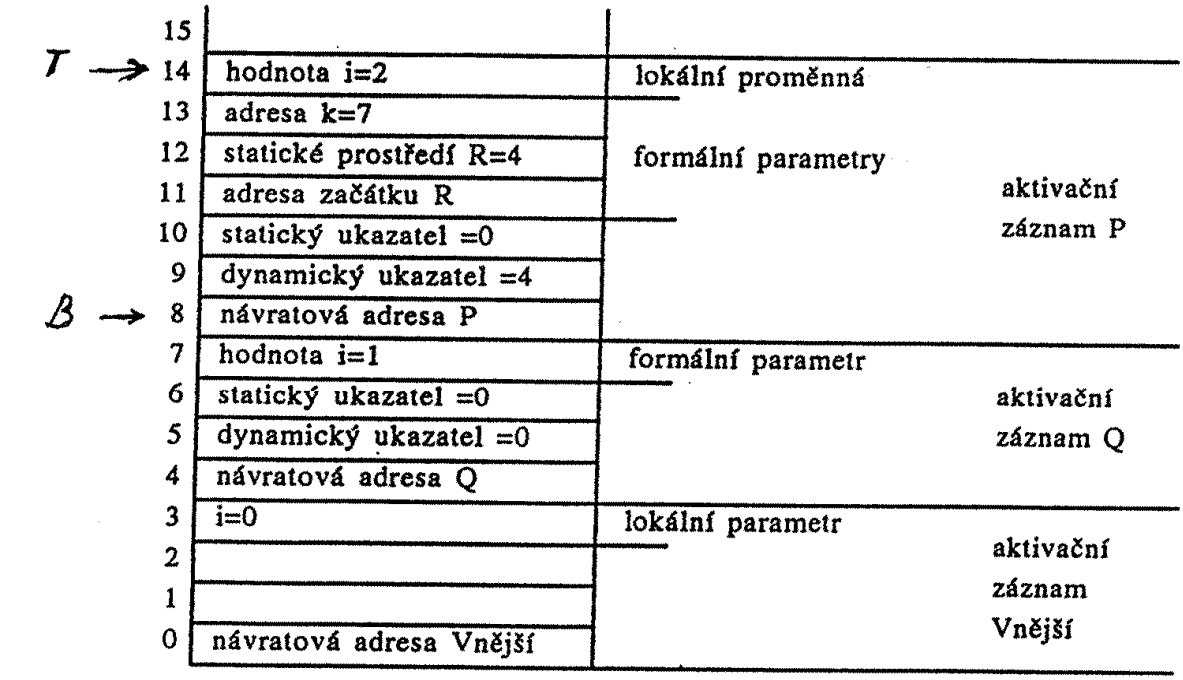
Při ad hoc vazbě se tiskne … ?

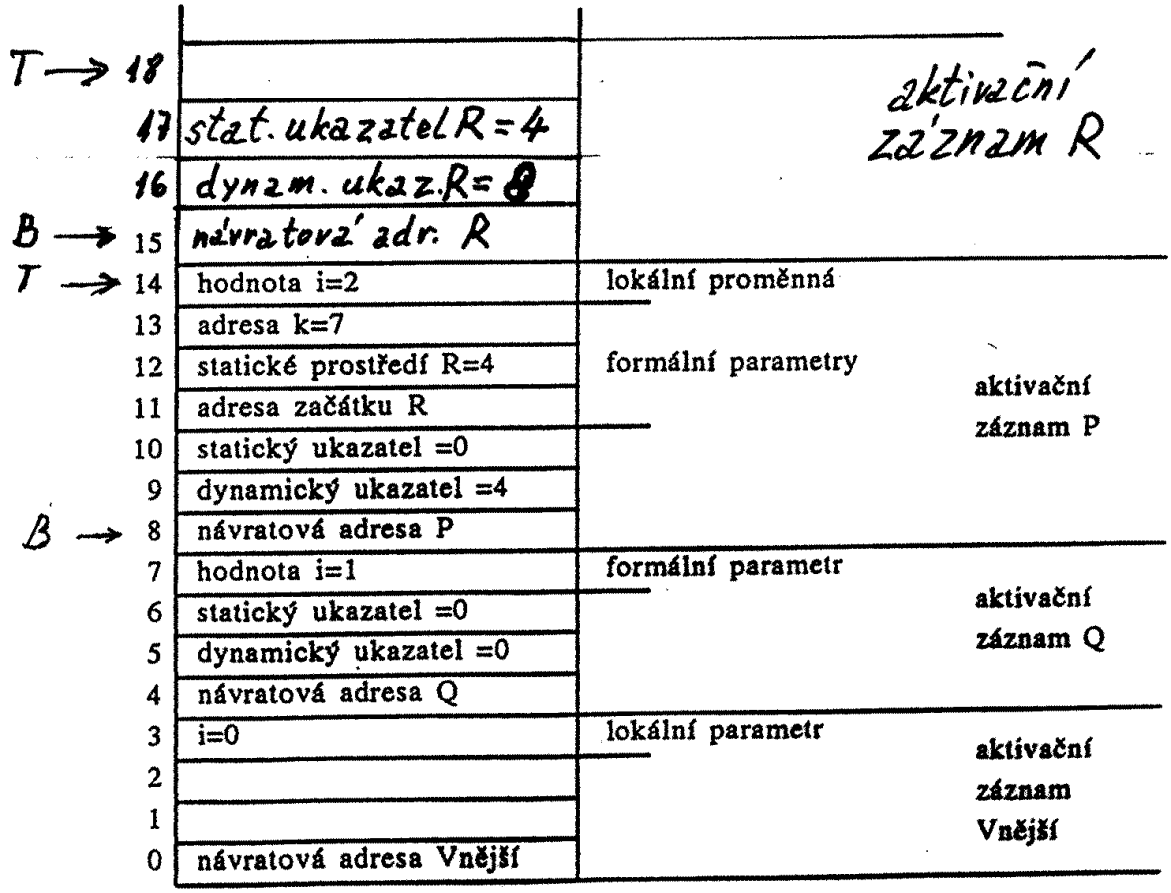
Př.5

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);
}
```

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





Přidělování paměti pro paralelní výpočty

Pro uložení AZ paralelního výpočtu nutno použít haldu nebo zobecněný zásobník

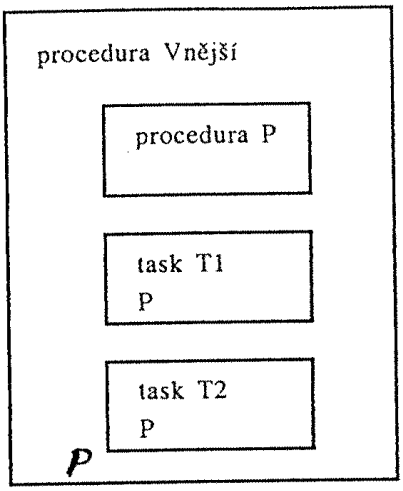
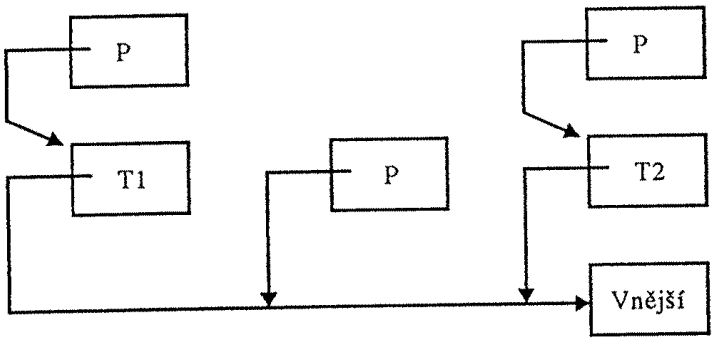
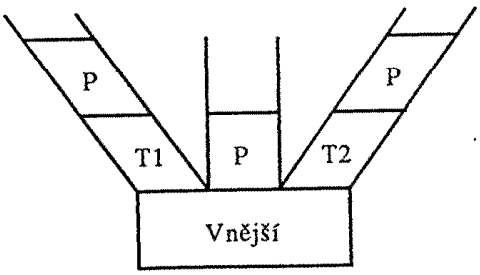


Schéma vnoření bloků programu

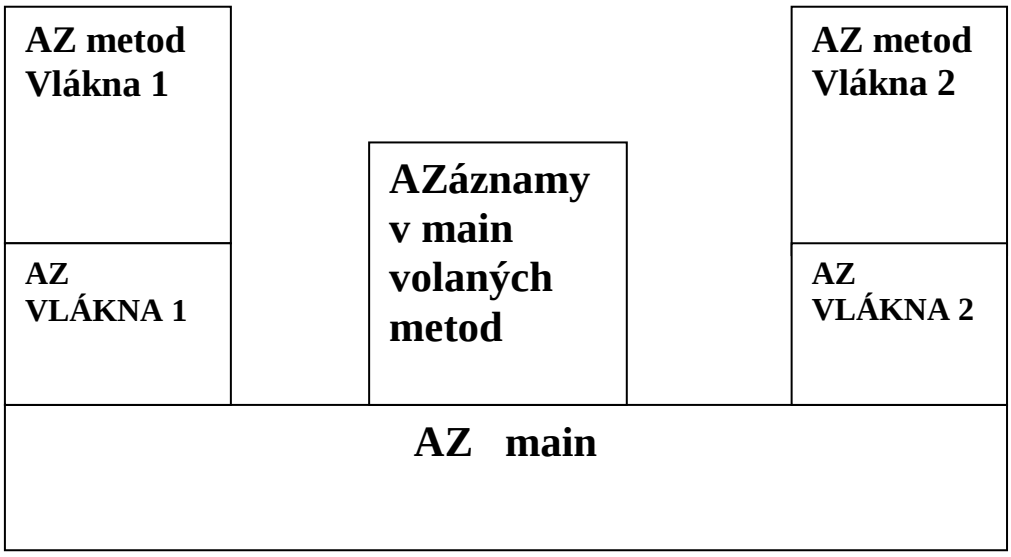


Struktura aktivačních záznamů při paralelním výpočtu



Uložení aktivačních záznamů ve zobecněném zásobníku

Př v javovském prostředí



Interpretace

Práce interpretu je obdobou práce základního cyklu procesoru

Registr instrukcí **RI**
Čítač instrukcí **PC**

Základní cyklus:

```
do {
   RI ← PROGRAM [ PC ];
   PC ← PC + 1;
   switch (RI.INSTRUCTION_CODE) {
      case INSTRUCTION_CODE_1: STATEMENTS_OF_INSTRUCTION_CODE_1;
      case INSTRUCTION_CODE_2: STATEMENTS_OF_INSTRUCTION_CODE_2;
                    :
      case INSTRUCTION_CODE_n: STATEMENTS_OF_INSTRUCTION_CODE_n;
   }
} while (RI.INSTRUCTION_CODE ≠ STOP);
```

Zavedme postfixové instrukce pro výpočty s integer

TA take address do zásob., parametrem je adresa (hladina, posun)
TC take constant do zásob., parametrem je hodnota
DR dereference vrcholu zásobníku,
ST store, obsah vrcholu ulož na adresu pod vrcholem,
JU jump, parametrem je adresa
IFJ if false jump, parametrem je adresa, vrchol je F=0 nebo T≠0
PLUS,
MINUS,
TIME,
DIV,
NEG změna znaménka,
AND,
OR,
NOT,
REL typ je dán parametrem (LT, LE, EQ, GE, GT, NE,
OD test lichosti,
READ čte do adresy na vrcholu zásobníku,
WRITE tiskne obsah vrcholu zásobníku ,
CSUB skok do podprogramu,
PAR parametrem je adresa skutečného parametru,
BBEG vstup do pp, vytvoří AZ, parametry hladina a velikost,
FPAR formální parametr, parametrem je VAR nebo CONST
RET, návrat z pp, likvidace jeho AZ
STOP konec výpočtu

```
Výpočtový zásobník je integer pole Z[ 1 .. MAXZ ];

Aktivační záznam bude mít tvar:

     T   ---->    pomocne promenne
                  Lokalni promenne
                  parametry                                  4
                  hladina                                    3
                  stara hodnota  DISPLAY [ level ]           2
                  dynamicky ukazatel                         1
     B   ---->    navratova adresa                           0



program INTERPRET;
 konstanty     MAXP = ...; /*max. délka programu*/
               MAXZ = ...; /*max. hloubka zásobníku*/
               MAXD = ...; /*max. velikost displeje*/
 typy  DPI = (TA, TC, DR, ST, JU IFJ, PLUS, MINUS,
              TIME, DIV, NEG, AND, OR, NOT, REL, OD, READ, WRITE,
              CSUB, PAR, BBEG, FPAR, RET, STOP);
       TPI = struktura (diskriminant IC typu DPI) {
         když IC je (DR, ST, PLUS, MINUS, TIME, DIV, NEG,
                AND, OR, NOT, OD, READ, WRITE, RET, STOP)pak  ();
         když IC je (TA, PAR) pak (N typu 1..MAXD;P typu 0..MAXZ);
         když IC je (TC)      pak (K typu integer);
         když IC je (JU, IFJ, CSUB) pak (I typu 0..MAXP);
         když IC je (REL)  pak (RO typu (LT, LE, EQ, GE, GT, NE));
         když IC je (BBEG) pak (H typu 1..MAXD; L typu integer);
         když IC je (FPAR) pak (V typu (CONST, VAR))
            };
 proměnné PROGRAMS typu array[0..MAXP] prvky typu TPI;
          Z        typu array[0..MAXZ] prvky typu integer;
          DISPLAY  typu array[1..MAXD] prvky typu 0..MAXZ;
          PC      typu 0..MAXP;
          B,T,TP  typu 0..MAXZ;
          RI      typu TPI;

 procedure READPROGRAM;
           /*načte postfixové instrukce do pole PROGRAMS*/

/*hlavni program*/
{  READROGRAM; T ← 0; PC ← 0; /* v PROGRAMS[0] je skok na první
                            vykonávanou instrukci    */
```

Programový text základního cyklu interpretu

```
}
```

```
do
 RI ← PROGRAMS[ PC ]; PC ← PC+1;
 switch RI.IC {
   case TA    :{ T←T+1; Z[T]←DISPLAY[RI.N] + RI.P };
   case TC    :{ T←T+1; Z[T]←RI.K };
   case DR    :Z[T] ← Z [ Z [ T ] ] ;
   case ST    :{ Z[Z[T-1]] ← Z[T]; T←T-2 };
   case JU    :PC ← RI.I;
   case IFJ   :{ if Z[T]=0 then PC←RI.I; T←T-1 }:
   case PLUS  :{ Z[T-1]←Z[T-1] + Z[T]; T←T-1; };
   case MINUS :{ Z[T-1]←Z[T-1] - Z[T]; T←T-1; };
   case TIME  :{ Z[T-1]←Z[T-1] * Z[T]; T←T-1; };
   case DIV   :{ Z[T-1]←Z[T-1] div Z[T]; T←T-1; };
   case NEG   :Z[T] ← -Z[T];
   case AND   :{ Z[T-1]←Z[T-1]*Z[T]; T←T-1 };
   case OR    :{ Z[T-1]←ord((Z[T-1]=1)or(Z[T]=1));T←T-1;};
   case NOT   :if Z[T]=0 then Z[T]←1 else Z[T]←0;
   case REL   :{ switch RI.RO {
                    case LT: Z[T-1] ← ord(Z[T-1]< Z[T]);
                    case LE: Z[T-1] ← ord(Z[T-1]<=Z[T]);
                    case EQ: Z[T-1] ← ord(Z[T-1]= Z[T]);
                    case GE: Z[T-1] ← ord(Z[T-1]>=Z[T]);
                    case GT: Z[T-1] ← ord(Z[T-1]> Z[T]);
                    case NE: Z[T-1] ← ord(Z[T-1]<>Z[T]);
                                 };
                 T ← T-1 ;
         };
   case OD    :Z[T] ← ord(odd(Z[T]));
   case READ  :{ read(Z[Z[T]]); T←T-1 };
   case WRITE :{ write(Z[T]); T←T-1 };
   case CSUB  :{ T←T+1; Z[T]←PC; PC←RI.I; TP←T+4 };
   case PAR   : ;
   case BBEG  :{ Z[T+1]←B; Z[T+2]←DISPLAY[RI.H]; B←T;
                 DISPLAY[RI.H]←B; Z[T+3]←RI.H; T←T+RI.L
         };
   case FPAR  :{ case RI.V of
                    VAR  :Z[TP]←DISPLAY[ PROGRAMS[ Z[ B ]].N]
                                + PROGRAMS[ Z[ B ] ].P;
                    CONST:Z[TP]←Z[ DISPLAY[ PROGRAMS[ Z[ B ]].N]
                                  +PROGRAMS[ Z[ B ]].P];
                 };
                 Z[ B ] ← Z[ B ] + 1; TP ← TP + 1;
         };
   case RET   :{ DISPLAY[Z[B+3]] ← Z[B+2]; T←B-1;
                 PC ← Z[ B ]; B ← Z[ B+1];
         };
   case STOP  : ;
 }
while RI.IC ≠ STOP;
```

Př.

```
{ /*program s rekurzivnim vnorenym podprogramem                    } Hladina 1
     integer m, n, k;

     podprogram NSD(integer i, j);

     {                                                             }
          while i <> j do                                          } hladina 2
               if i > j  then i =  i - j                           }
                         else j = j – i;                           }
          k = i;                                                   }
     };

     read(m); read(n);                                             }
     if (m > 0) and (n > 0) then                                   } Hladina1
                              { NSD(m, n); write(k);               }
                              }                                    }
}
```

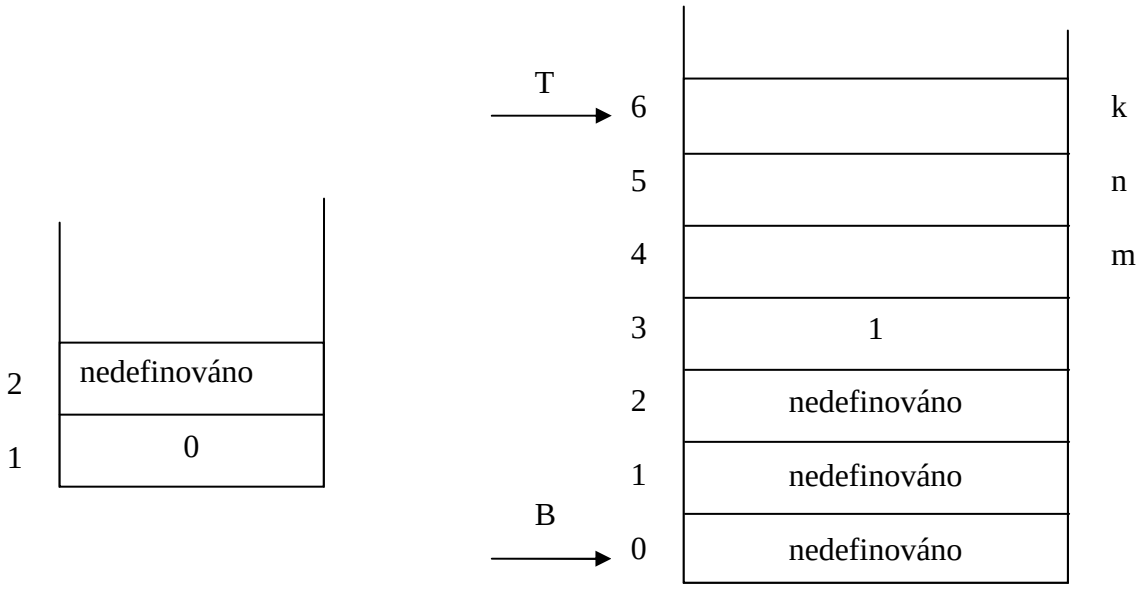
Jméno proměnné	hladina	posun
m	1	4
n	1	5
k	1	6
i	2	4
j	2	5

Program přeložený do postfixových instrukcí

```
(0)  JU   37
(1)  BBEG 2,5   začátek deklarace
(2)  FPAR CONST   procedury NSD
(3)  FPAR CONST
(4)  TA   2,4     začátek while
(5)  DR
(6)  TA   2,5     i ≠ j
(7)  DR
(8)  REL  NE
(9)  IFJ  32      výskok while
(10) TA   2,4     začátek if
(11) DR
(12) TA   2,5     i > j
(13) DR
(14) REL  GT
(15) IFJ  24
(16) TA   2,4
(17) TA   2,4
(18) DR
(19) TA   2,5     i ← i - j
(20) DR
(21) MINUS
(22) ST
(23) JU   31
(24) TA   2,5
(25) TA   2,5
(26) DR
(27) TA   2,4     j ←j-i
(28) DR
(29) MINUS
(30) ST           konec if
(31) JU   4       konec while
(32) TA   1,6
(33) TA   2,4      k ←i
(34) DR
(35) ST           konec dekl.
(36) RET          proc.NSD
(37) BBEG 1,6
(38) TA   1,4
(39) READ         read(m)
(40) TA   1,5
(41) READ         read(n)
(42) TA   1,4     začátek if
(43) DR
(44) TC   0       m > 0
(45) REL  GT
(46) TA   1,5
(47) DR
(48) TC   0       n > 0
(49) REL  GT
(50) AND
(51) IFJ  58
(52) CSUB 1
(53) PAR  1,4     NSD(m,n)
(54) PAR  1,5
(55) TA   1,6
(56) DR           write(k)
(57) WRITE        konec if
(58) STOP
```

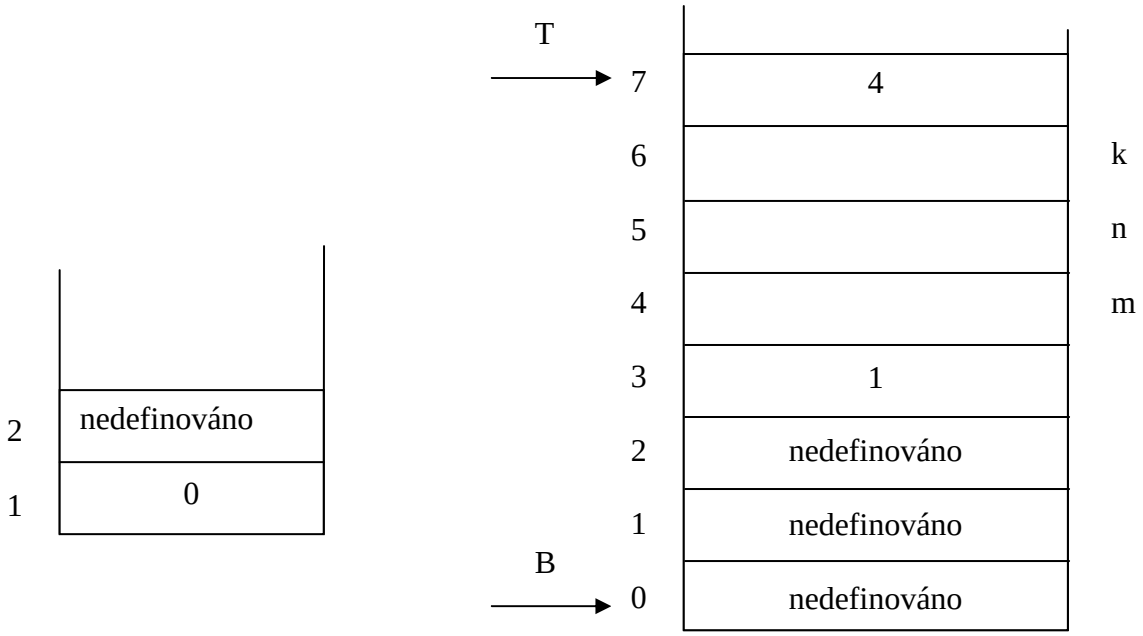
Budeme předpokládat čtení hodnot 60 a 90

T = 0 PC = 0
Stav po provedení instrukce (37) BBEG 1,6



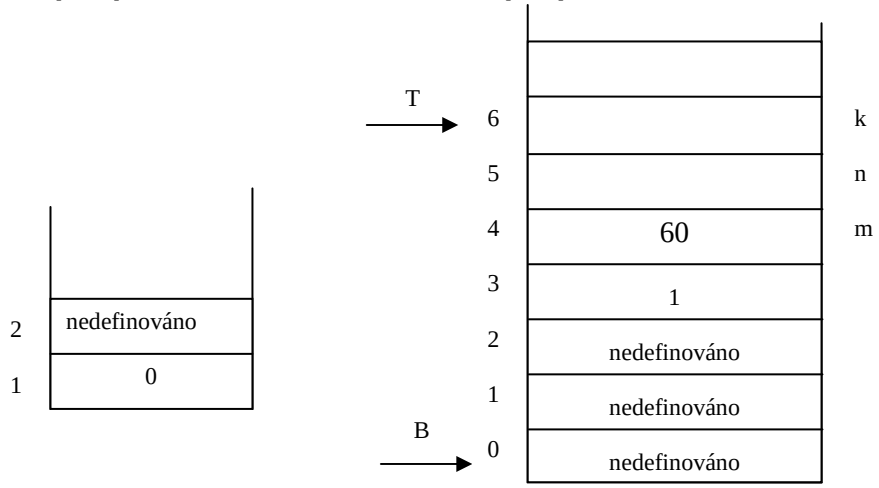
DISPLAY **Zásobník**

Stav po provedení instrukce (38) TA 1,4



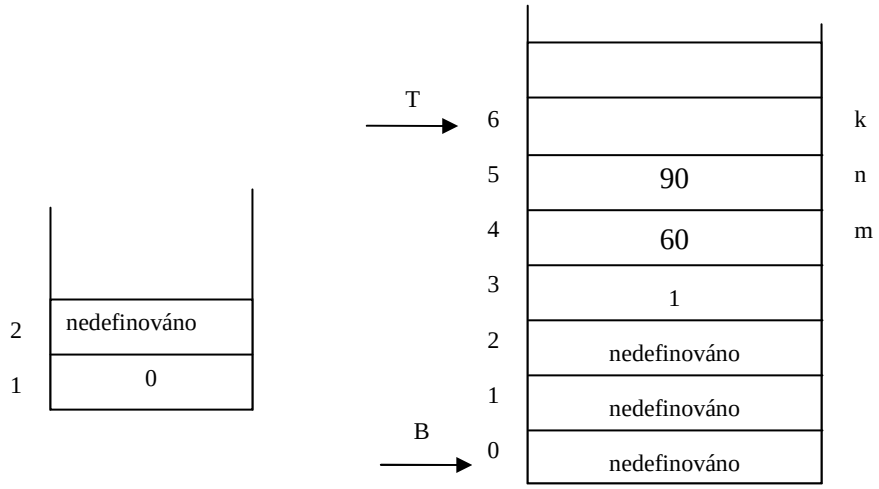
DISPLAY **Zásobník**

Stav po provedení instrukce (39) READ



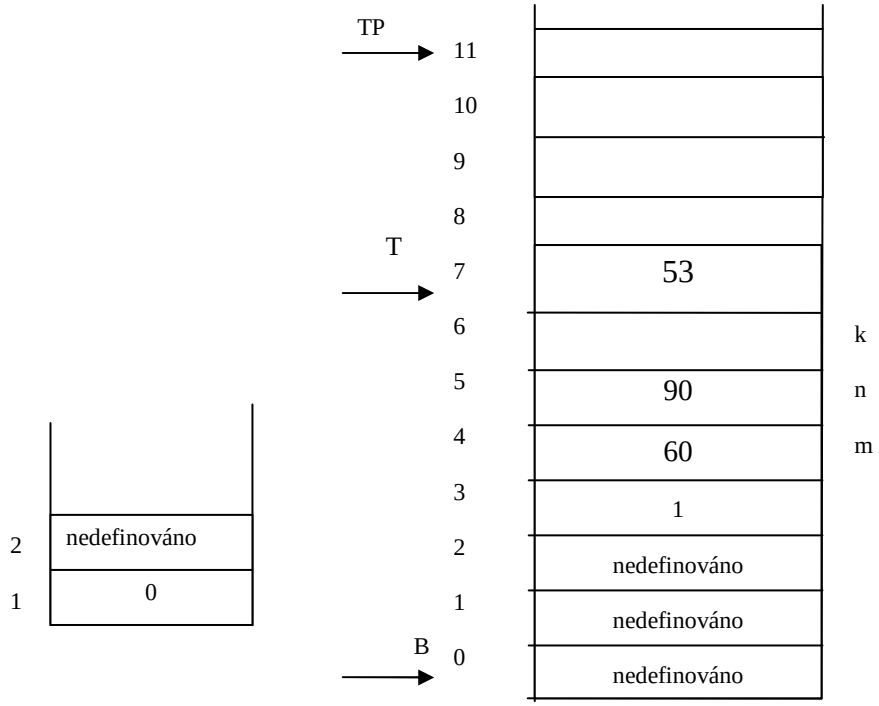
DISPLAY **Zásobník**

Stav po provedení instrukce (40) TA 1, 5
(41) READ



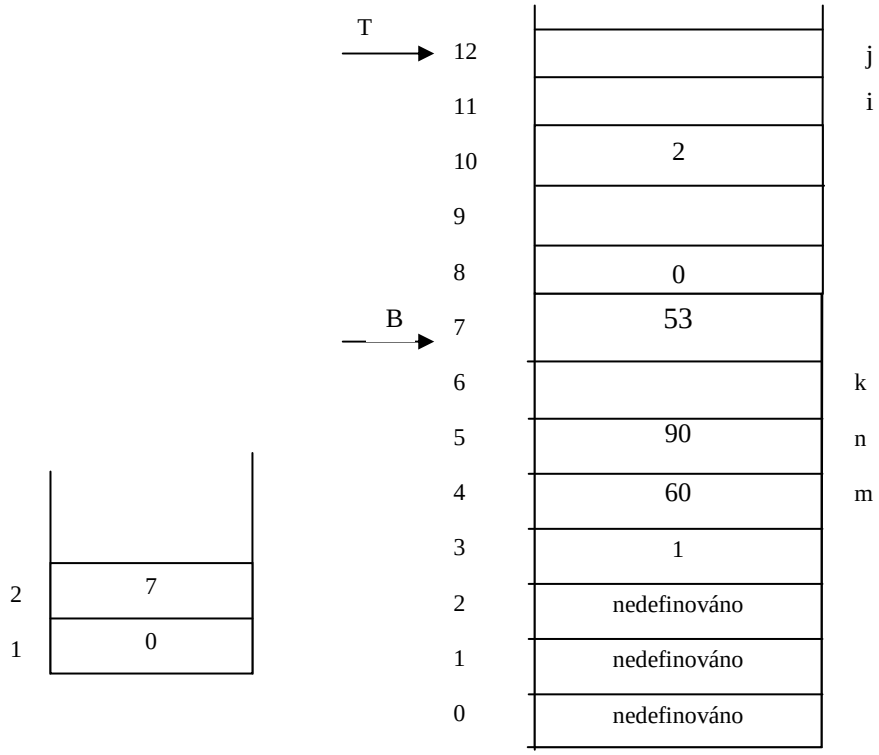
DISPLAY **Zásobník**

Stav po provedení instrukce (52) CSUB 1



PC = 1

Stav po provedení instrukce (1) BBEG 2, 5



Zpracování dalších instrukcí (z podprogramu NSD)

Instrukce	Změna v Z	Další změny
(2) FPAR CONST	$Z[11] := 60$ $Z[7] := 54$	$TP := 12$
(3) FPAR CONST	$Z[12] := 90$ $Z[7] := 55$	$TP := 13$
(4) TA 2,4	$Z[13] := 11$	$T := 13$
(5) DR	$Z[13] := 60$	
(6) TA 2,5	$Z[14] := 12$	$T := 14$
(7) DR	$Z[14] := 90$	
(8) REL NE	$Z[13] := 1$	$T := 13$
(9) IFJ 32		$T := 12$
(10) TA 2,4	$Z[13] := 11$	$T := 13$
(11) DR	$Z[13] := 60$	
(12) TA 2,5	$Z[14] := 12$	$T := 14$
(13) DR	$Z[14] := 90$	
(14) REL GT	$Z[13] := 0$	$T := 13$
(15) IFJ 24		$T := 12$ $PC := 24$
(24) TA 2,5	$Z[13] := 12$	$T := 13$
(25) TA 2,5	$Z[14] := 12$	$T := 14$
(26) DR	$Z[14] := 90$	
(27) TA 2,4	$Z[15] := 11$	$T := 15$
(28) DR	$Z[15] := 60$	
(29) MINUS	$Z[14] := 30$	$T := 14$
(30) ST	$Z[12] := 30$	$T := 12$
(31) JU 4		$PC := 4$
(4) TA 2,4	$Z[13] := 11$	$T := 13$
(5) DR	$Z[13] := 60$	
(6) TA 2,5	$Z[14] := 12$	$T := 14$
(7) DR	$Z[14] := 30$	

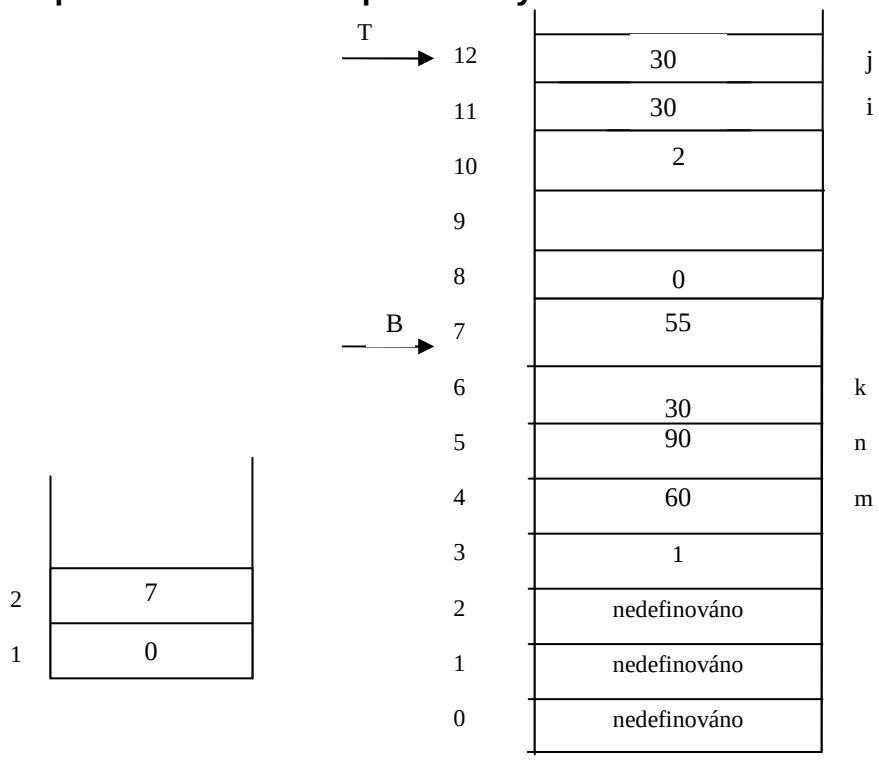
Instrukce	Změna v Z	Další změny
(8) REL NE	$Z[13] := 1$	$T := 13$
(9) IFJ 32		$T := 12$
(10) TA 2,4	$Z[13] := 11$	$T := 13$
(11) DR	$Z[13] := 60$	
(12) TA 2,5	$Z[14] := 12$	$T := 14$
(13) DR	$Z[14] := 30$	
(14) REL GT	$Z[13] := 1$	$T := 13$
(15) IFJ 24		$T := 12$
(16) TA 2,4	$Z[13] := 11$	$T := 13$
(17) TA 2,4	$Z[14] := 11$	$T := 14$
(18) DR	$Z[14] := 60$	
(19) TA 2,5	$Z[15] := 12$	$T := 15$
(20) DR	$Z[15] := 30$	
(21) MINUS	$Z[14] := 30$	$T := 14$
(22) ST	$Z[11] := 30$	$T := 12$
(23) JU 31		$PC := 31$
(31) JU 4		$PC := 4$
(4) TA 2,4	$Z[13] := 11$	$T := 13$
(5) DR	$Z[13] := 30$	
(6) TA 2,5	$Z[14] := 12$	$T := 14$
(7) DR	$Z[14] := 30$	
(8) REL NE	$Z[13] := 0$	$T := 13$
(9) IFJ 32		$T := 12$ $PC := 32$
(32) TA 1,6	$Z[13] := 6$	$T := 13$
(33) TA 2,4	$Z[14] := 11$	$T := 14$
(34) DR	$Z[14] := 30$	
(35) ST	$Z[6] := 30$	$T := 12$

Tabulka . Další provádění programu

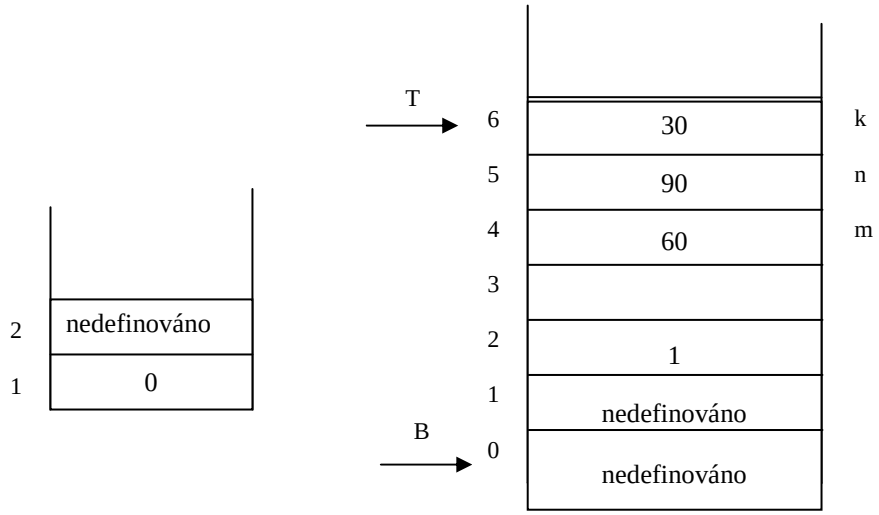
Instrukce	Změna v Z	Další změny
(55) TA 1,6	$Z[7] := 6$	$T := 7$
(56) DR	$Z[7] := 30$	
(57) WRITE		$T := 6$; tiskne hodnotu 30
(58) STOP		ukončí interpretaci

Instrukce ukončující interpretaci

Stav před návratem z procedury



Stav po provedení instrukce (36) RET



Interpretace v PL0

Instrukce postfixového zápisu

lit 0,A **uloz konstantu A do zasobniku**

opr 0,A **proved instrukci A**

1	**unarni minus**
2	**+**
3	**-**
4	*****
5	**div**
6	**mod**
7	**odd**
8	**=**
9	**<>**
10	**<**
11	**>=**
12	**>**
13	**<=**

lod L,A **:ulož hodnotu proměnné z adr. L,A na vrchol zásobníku**

sto L,A **:zapiš do proměnné s adr. L,A hodnotu z vrcholu zásob.**

cal L,A **:volej proceduru s adresou kódu A z úrovně L**

ret 0,0 **:return**

ing 0,A **:zvyš obsah Top-registru zásobníku o hodnotu A**

jmp 0,A **:proveď skok na adresu kódu A**

jpc 0,A **:proveď podmíněný skok na adresu kódu A**

```
/* interpretace generovanych kodu PL0. S je vypoctovy zasobnik, program je v code[ ]*/
void interpret(void) {
int p, t;                   /* citac instrukci,  vrchol zasobniku */
INSTRUCTION i;      /*registr instrikce*/

   printf("START PL/0\n");
   t = p = s[1] = s[2] = s[3] = 0;  /*vrchol, citac, stat.retez, dynamo.retez, navrat.adresa*/
   b = 1;
   do {
      i = code[p++];

      switch (i.f) {
         case lit: s[++t] = i.a;
                  break;
         case opr:
            switch (i.a) {
               case neg : s[t] = -s[t];
                                       /*printf("INTERPRET OPR: neg\n");*/
                                       break;
               case add : t--;
                        s[t] += s[t + 1];
                                       /*printf("INTERPRET OPR: add\n");*/
                                       break;
               case sub : t--;
                                       s[t] -= s[t + 1];
                                       /*printf("INTERPRET OPR: sub\n");*/
                                       break;
               case mul : t--;
                                       s[t] *= s[t + 1];
                                       /*printf("INTERPRET OPR: mul\n");*/
                                       break;
               case di :  t--;
                                       if (s[t + 1] != 0) s[t] /= s[t + 1];
                        else {
                           // error
                           error(31);
                        }
                        /*printf("INTERPRET OPR: di\n");*/
                        break;
               case mod : t--;
                                       s[t] = s[t] % s[t + 1];
                                       /*printf("INTERPRET OPR: mod\n");*/
                                       break;
               case odd : s[t] = s[t] % 2;
                                       /*printf("INTERPRET OPR: odd\n");*/
                                       break;
               case eq : t--;
                                 s[t] = (s[t] == s[t + 1]);
                                       /*printf("INTERPRET OPR: eq\n");*/
                        break;
               case ne : t--;
                                 s[t] = (s[t] != s[t + 1]);
                                       /*printf("INTERPRET OPR: ne\n");*/
                        break;
               case lt : t--;
```

```
                        s[t] = (s[t] < s[t + 1]);
                            /*printf("INTERPRET OPR: lt\n");*/
                    break;
                case ge : t--;
                        s[t] = (s[t] >= s[t + 1]);
                            /*printf("INTERPRET OPR: ge\n");*/
                        break;
                case gt : t--;
                        s[t] = (s[t] > s[t + 1]);
                            /*printf("INTERPRET OPR: gt\n");*/
                    break;
                case le : t--;
                        s[t] = (s[t] <= s[t + 1]);
                            /*printf("INTERPRET OPR: le\n");*/
                        break;

                }
                break;

        case lod: t++;   /*natazeni adresy promenne do stacku*/
                    s[t] = s[base(i.l) + i.a];        /*fce base provede sestup o l urovni po stat.retezu*/
                    break;        /* v PL0 je dynam.adresa (hlad.pouziti minus hlad.deklarace, posuv)

        case sto: s[base(i.l) + i.a] = s[t];
                    printf("%d\n",s[t--]);
                    break;

        case cal: s[t + 1] = base(i.l);     /*staticky retezec*/
                    s[t + 2] = b;           /*dynamicky retezec*/
                    s[t + 3] = p;           /*navratova adresa*/
                    b = t + 1;              /*nova baze*/
                    p = i.a;                /*zacatek podprogramu*/
                break;

        case ret: t = b - 1;
                    p = s[t + 3];           /*do p dame navratovou adresu*/
                    b = s[t + 2];           /*nastavime starou bazi*/
                    break;

        case ing: t += i.a;         /*a je velikost AZ = 3+pocet promennych*/
                break;

        case jmp: p = i.a;
                break;

        case jpc: if (s[t] == 0) p = i.a;   /*skok při false*/
                    t--;
                    break;

    }
  } while (p);
  printf(" END PL/0\n");

}   // interpret()
```

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 j e 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 (pohodáři jej najdou na konci)

2. Generování z trojic popíšeme rozhodovací tabulkou COMP

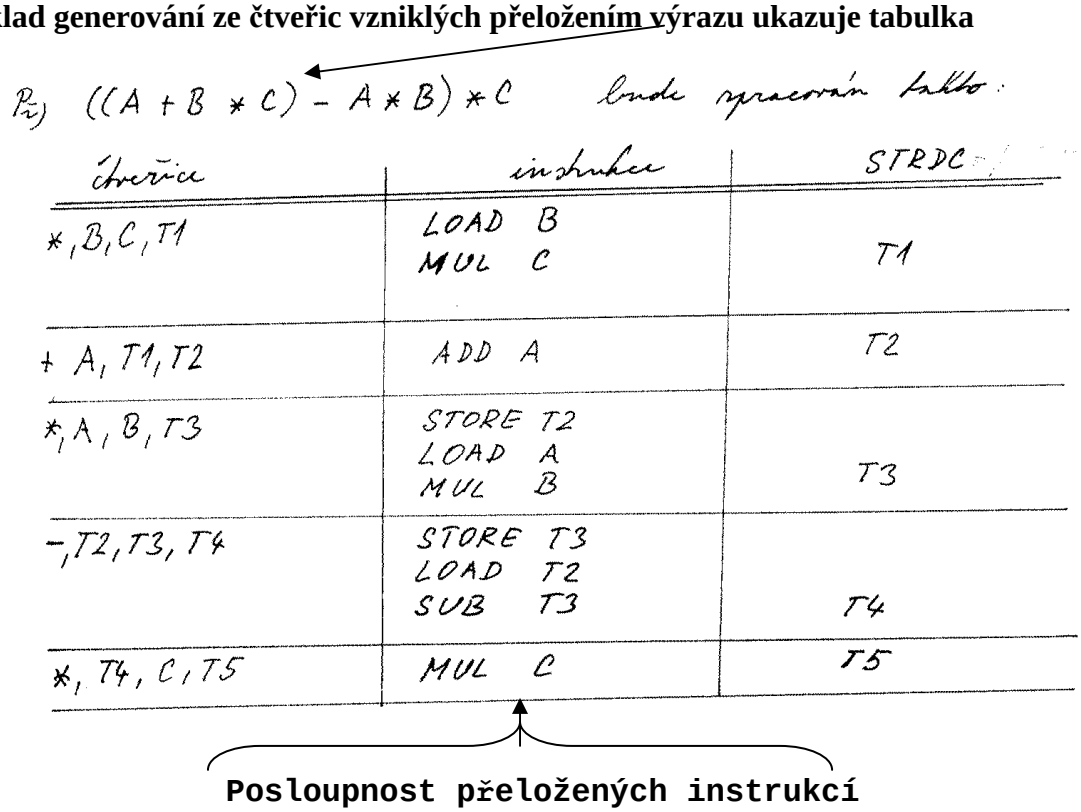
operator	op1 \ op2	accumulator	proměnná	trojice
+	accumulator		GEN('ADD',OP2)	T← NTV GEN('STORE',T) COMP(OP2) GEN('ADD',T)
	proměnná	GEN('ADD',OP1)	GEN('LOAD',OP1) GEN('ADD',OP2)	COMP(OP2) GEN('ADD',OP1)
	trojice		COMP(OP1) GEN('ADD',OP2)	COMP(OP1) OP1← accumulator COMP(Self)
-	accumulator		GEN('SUB',OP2)	
	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 (pohodáři jej najdou na posl.str.)

Příklad generování ze čtveřic vzniklých přeložením výrazu ukazuje tabulka



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 ni specifikovaných akcí pro konkrétní trojice se snaží zachytit následující obr.

Př.
A*(B+C)-B*(A+C) přeloženo do trojic má tvar
(1): +,B,C
(2) *,A,(1)
(3): +,A,C
(4): *,B,(3)
(5): -,(2),(4)
Generování začne vždy od poslední trojice

(5): -,(2),(4)
Volá KOMP(4) (4): *,B,(3)
Volá KOMP(3) (3): +,A,C
GEN(„LOAD“, A)
GEN(„LOAD“, C)
Návrat do (4)
GEN(„MUL“,B)
Návrat do (5)
OP2 ←**accumulator**, tzn modifikuje (5)
Volá KOMP(modif.5) (mod 5): -,(2), accumulator
T_1 ← **NTV** **vytvoří novou pomocnou proměnnou**
GEN(„STORE“, T_1 **)**
Volá KOMP(2): (2) *,A,(1)
Volá KOMP(1) (1): +,B,C
GEN(„LOAD“,B)
GEN(„ADD“,C)
Návrat do (2)
GEN(„MUL“,A)
Návrat do (mod 5)
GEN(„SUB“, T_1**)**
Návrat z (mod5) do (5)
Konec

Bezkontextové gramatiky

Def. BKG: G = (N, T, P, S) kde

1. **N je množina neterminálních symbolů,**
2. **T " " terminálních " ,**
3. **S ∈ N je počáteční symbol,**
4. **P je množina přepisovacích pravidel tvaru**

A -> α , kde A ∈ N, α ∈ (N ∪ T)*

Bezkontextový jazyk

Def. BKL: L(G) = { w : S =>* w , w ∈ T* }

Tj.L(G) je množina řetězců derivovatelných z S

Úmluva pro zjednodušení zápisů:

a, b, c, ... představují terminální symboly
A, B, C, ... „ neterminální „
X, Y, Z, ... „ N ∪ T
α, β, γ, ... „ řetězce z N ∪ T
u, v, z, . . „ „ z terminálních symbolů
e představuje prázdný řetězec

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

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

Dtto S =>* α pozn.: =>* je uzávěr relace =>

- PŘÍMÁ DERIVACE $\alpha A \beta \Rightarrow \alpha \gamma \beta$, **kde** $A \to \gamma \in P$

- DERIVAČNÍ STROM je grafickým vyjádřením derivace (struktury) řetězce. Kořenem je počáteční symbol, uzly jsou prvky N ∪ T, listy jsou prvky T, větve z uzlu A vedou do uzlů, které zleva doprava tvoří řetězec α, který je pravou stranou pravidla A → α

Př. G[E]

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

Vytvořte derivační strom a derivaci věty např. i + i * i (na tabuli)

Vztah derivace a derivačního stromu: derivaci odpovídá jeden strom, jednomu stromu odpovídá více derivací.

- **KANONICKÉ DERIVACE**
  - **Levá derivace -expanduje vždy nejlevější neterminál**
  - **Pravá derivace -expanduje vždy nejpravější neterminál**

Př. levá a pravá derivace věty i + i * i
na tabuli

- VĚTNÁ FORMA

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

$$S \Rightarrow^* \alpha, \text{ kde } \alpha \in (N \cup T)^*$$

- VĚTA

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

$$S \Rightarrow^* \alpha, \text{ kde } \alpha \in T^*$$

- FRÁZE

Def.: Nechť $\lambda = \alpha\ \beta\ \gamma$ je větná forma v gramatice G. Podřetězec β se nazývá frází větné formy λ vzhledem k neterminálnímu symbolu A, platí-li

$$S \Rightarrow^* \alpha A \gamma \quad a \quad A \Rightarrow^* \beta$$

Tzn.frázi tvoří listy podstromu derivačního stromu.

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

$$S \Rightarrow^* \alpha A \gamma \quad a \quad A \Rightarrow \beta$$

- L-FRÁZE

je nejlevější jednoduchou frází

Př.
Najdi fráze, jednoduché fráze a l-frázi větné formy i*i+i v G[E] (na tabuli)

Problémy analýzy při konstrukci derivačního stromu:

1.(shora dolů) Kterou z pravých stran vybrat k derivování

2.(zdola nahoru) Jak vymezit l-frázi a na co ji redukovat

řešení: -buď analýza s návratem (neefektivní, složitost kubická)

-nebo deterministická analýza (jen pro některé, druhy BKG)

Víceznačnost gramatik

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

Př. Jazyk { $a^m c a^n$; m, n >= 0 }

je generován gramatikou S -> aS | Sa | c

-Je věta aaca jednoznačná? jak vypadá strom, je jen jeden?

-Může pro nejednoznačnou gramatiku existovat ekvivalentní jednoznačná gramatika? Může: S -> aS | Z
Z -> Za | c

Př. G[E] E -> E + E | E * E | i

-Jaké jsou důsledky v generovaném jazyce ?

Věta: 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. Tzn. je dokázáno, že nikdy nebude existovat pro takový problém algoritmus.

Př. Syntaktický tvar podmíněného příkazu:

S -> a S b S | a S | c

-Je G[S] víceznačná ?

S1 -> a S2 b S1 | a S1 | c
S2 -> a S2 b S2 | c

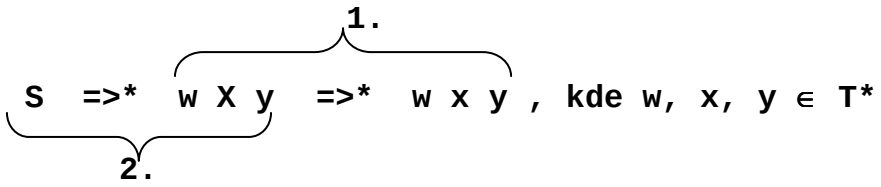
Gramatika je také, víceznačná, existují-li v G pro rekurzivní neterm. symbol A alespoň 2 rekurzivní pravidla, z nichž jedno je rekurzivní zprava (zleva) a má shodný prefix (postfix) rekurzivního symbolu A s druhým pravidlem.

Jazyky, které nelze generovat jednoznačnou gramatikou se nazývají inherentně nejednoznačné.

Úpravy gramatik

Odstranění zbytečných symbolů

Zbytečný je takový symbol X, který buď (1.)je-li neterminální z něj nelze generovat terminální řetězec, nebo (2.)ať je terminální či neterminální, je nedosažitelný z S.



Postup při eliminaci zbytečných symbolů

1. **a)Označíme všechny X∈T.**
   b)Označíme všechny X∈N, pro něž existuje X-pravidlo, jehož pravá strana neobsahuje neoznačený symbol.
   c)Opakujeme krok b), dokud přibývá označených symbolů.
   d)Neoznaznačené symboly jsou zbytečné.
2. **a) Označíme počáteční symbol S.**
   b) Označíme všechny symboly z pravých stran pravidel s označeným levostranným symbolem.
   c)Opakujeme krok b), dokud přibývá označených symbolů.
   d)Neoznaznačené symboly jsou zbytečné.

! záleží na pořadí kroků 1. a 2. !

Př. G[S]: S → a | A A → A B B → b

Odstranění prázdných pravidel

Gramatika G je bez prázdných pravidel, jestliže buď neobsahuje žádné pravidlo A → e, nebo obsahuje jediné takové pravidlo tvaru S → e a S se nevyskytuje na pravé straně žádného pravidla v G.

Postup při odstranění prázdných pravidel

1. **Označíme všechny symboly X, pro něž existuje pravidlo s prázdnou pravou stranou.**
2. **Označíme všechny symboly X, pro něž existuje pravidlo s pravou stranou obsahující pouze označené symboly.**
3. **Opakujeme 2 dokud přibývá označených symbolů.**
4. **Takto získanou množinu označíme N_e .**
5. **Každé pravidlo gramatiky mající na pravé straně jeden či více symbolů z N_e , nahradíme množinou pravidel vzniklých všemi možnými způsoby vypuštění v pravých stranách symbolů z N_e. Případně vznikající pravidla tvaru X → e do výsledné gramatiky nezařazujeme.**
6. **Obsahuje-li N_e počáteční symbol S, vytvoříme nový počáteční symbol S´ s pravidly S´ → e a S´ → S**

(Gramatika bez prázdných pravidel je nezkracující, větné formy při derivování se nezkracují)

Př. Na tabuli. Odstraňte prázdná pravidla z G[S]: S → a S b S | e

Výsledek: S´ → S | e
S → a S b S | a b S | a S b | a b

Odstranění jednoduchých pravidel

Jednoduchá pravidla mají tvar A → B , kde A, B ∈ N

Odstranění = žádný problém = nahradíme A → B všemi možnými pravidly vzniklými záměnou B za pravé strany B-pravidel
Př. Zkusme pro G[E] na tabuli

Odstranění cyklů

A =>* A implikuje existenci jednoduchých pravidel
Cyklus je evidentní nešvar. Proč?
Cykly eliminujeme odstraněním jednoduchých pravidel.

Odstranění libovolného pravidla

Nechť chceme z G odstranit pravidlo A → α B β. Musíme proto místo něj dát do G všechna pravidla tvaru A → α γ β, kde γ jsou pravé strany B pravidel.

Př. V G s pravidly A → a A A | b odtranit pravidlo A → a A A
Na tabuli
Výsledek: A → a a A A A | a b A | b

Upravená gramatika

neobsahuje cykly, e-pravidla a zbytečné symboly

Odstranění levé rekurze

(Greibachové normální forma: ∀pravé strany začínají terminálem)

Levorekurzivní gramatiku nelze použít k analýze shora dolů

Odstranění pravidla rekurzivního zleva:

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

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

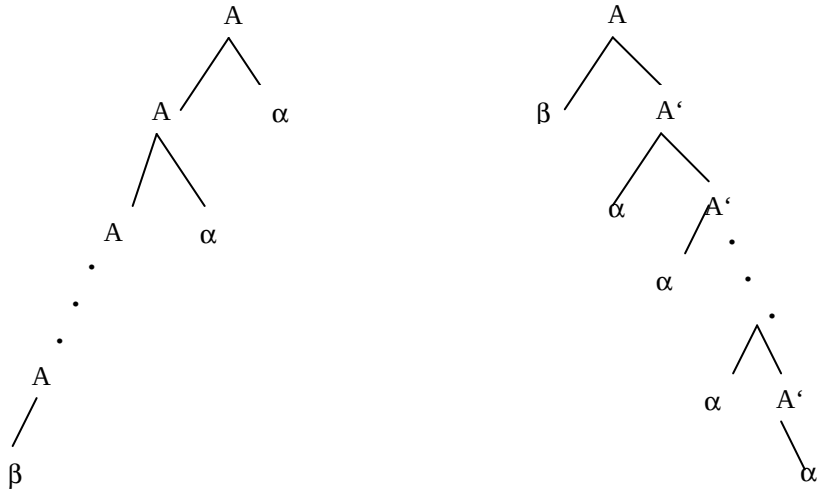
jsou všechna A pravidla v P a žádné, z β nezačíná A.

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

$$A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \mid \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A'$$

je ekvivalentní s gramatikou G

Levou rekurzi nahradíme pravou, jak je zřejmé z obr.

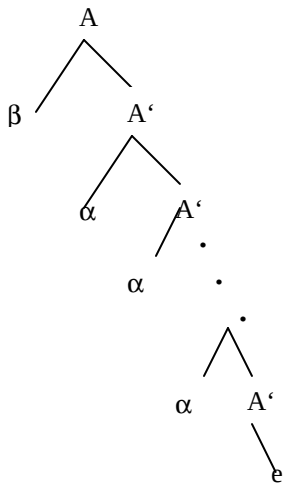


Př.a) G[E] na tabuli. E → E + T | T **?proč nepoužijeme E → T + E**
T → T * F | F
F → (E) | i

Alternativa odstranění s kratším výsledkem:

Ekvivalentní bude jak vidno z obr. i gramatika s pravidly

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid e$$



Př.b) G[E] na tabuli.

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid i$$

Pro pohodlné:

Výsledek př.a)

$$E \rightarrow T \mid T\ E'$$
$$E' \rightarrow +T \mid +\ T\ E'$$
$$T \rightarrow F \mid F\ T'$$
$$T' \rightarrow *\ F \mid *\ F\ T'$$
$$F \rightarrow (\ E\ ) \mid i$$

výsledek př.b)

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E' \mid e$$
$$T \rightarrow F\ T'$$
$$T' \rightarrow *\ F\ T' \mid e$$
$$F \rightarrow (\ E\ ) \mid i$$

Odstranění levé rekurze (včetně nepřímé rekurze):

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

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

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

   Jinak přiřaď i = i + 1 ; j = 1

4. Každé, pravidlo tvaru Ai → Aj γ nahraď pravidly

   Ai → α_1 γ | α_2 γ| … | α_p γ, kde

   Aj → α_1 | α_2 | … | α_p jsou všechna Aj pravidla

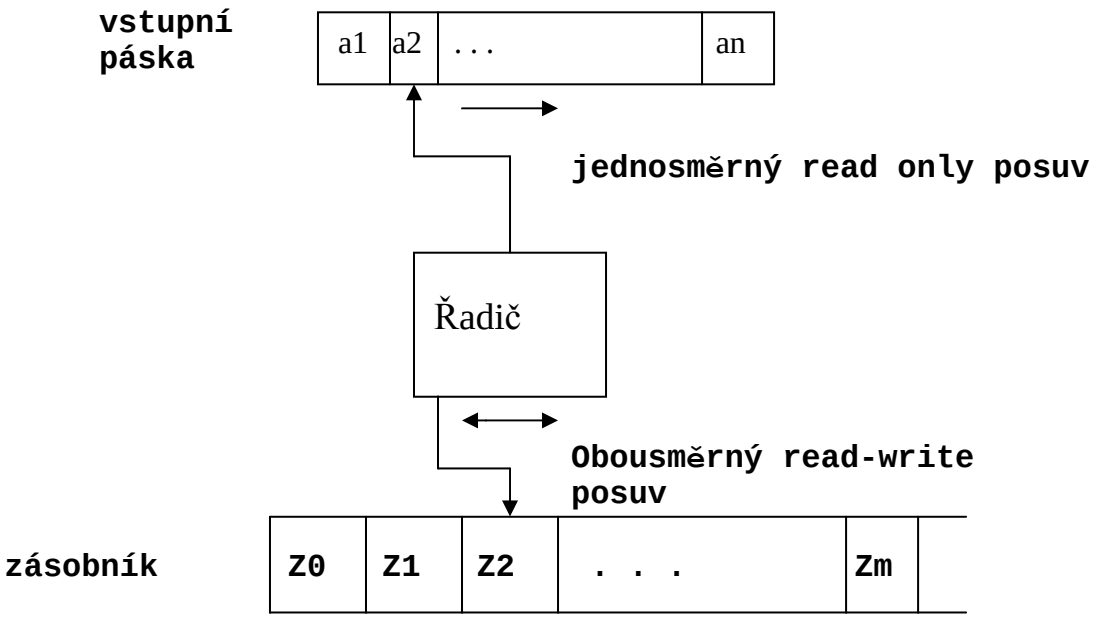
5. Je-li j = i - 1 jdi na krok 2., jinak j = j + 1 a jdi na 4.

Př. na tabuli s použitím kratší alt.: A → B C | a
B → C A | A b
C → A B | CC | a

1. Zvolíme uspořádání A_1 = A < A_2 = B < A_3 = C a přiřadíme i=1
2. Odstanime případnou levou rekurzi u A_1 pravidel(žádná tam není)
   Proto do výsledku jdou pravidla A → B C | a
3. i ≠ n, proto i=2 a j=1
4. Vnutíme uspořádání B pravidlům: B → C A | B C b | a b
5. j = i-1 proto dělej bod 2.
2. Teď z nich odstraníme přímou levou rekurzi. Do výsledku jde:
   B → C A B' | a b B' B' → C b B' | e
3. i ≠ n, proto i=3 a j=1
4. Vnutíme uspořádání C pravidlům s ohledem na A:
   C → B C B | a B | CC | a
5. j ≠ i-1 proto j = 2 a dělej znovu bod 4.
4. Vnutíme uspořádání C pravidlům i vzhledem k B:
   C → C A B'C B | a b B' C B | a B | CC | a
5. j = i-1 proto dělej bod 2 a do výsledku půjde:
2. C → a b B' C B C'| a B C'| a C'
   C'→ A B'C B C'| C C'| e
3. Konec

Zásobníkové automaty

ZA je abstraktní model syntaktického analyzátoru BK jazyků
Obecně je:
-jednocestný,
-nedeterministický,
-s nekonečnou pamětí (zásobníkem)



Definice:
ZA $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$
1 2 3 4 5 6 7
1.Konečná množina stavů
2.Konečná vstupní abeceda
3.Konečná abeceda zásobníkových symbolů
4.Zobrazení δ: $Q \times (\Sigma \cup \{e\}) \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$
5.Počáteční stav řadiče $\in Q$
6.Dno zásobníku $\in \Gamma$
7.Množina koncových stavů $\subset Q$

Konfigurace ZA $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$

Stav řadiče
Dosud nepřečtený vstup
Obsah zásobníku

Pozn.: Je to akceptační automat, ne překladový.
Přechod ZA je binární relace $\vdash$ **nad množinou konfigurací,**
nebo její p-tou mocninou $\vdash_p$ **či uzávěrem** $\vdash_*$ **a** $\vdash_+$
$(q, aw, \alpha\beta) \vdash (p, w, \gamma\beta)$ **jestliže** $\delta(q, a, \alpha)$ **obsahuje** (p, γ), $a \in T \cup \{e\}$, $\alpha, \beta, \gamma \in \Gamma^*$
Př. na tabuli. Popsat $\mathcal{P}$ **akceptující** $L = \{ 0^n 1^n \}$ **kde** $n \geq 0$

Počáteční konfigurace ZA je (q_0, w, Z_0), kde $w \in \Sigma^*$

Interpretace zápisu přechodové fce
$\delta(q, a, b) = \{ (p1, \gamma1), (p2, \gamma2), \ldots, (pn, \gamma n) \}$
ZA ve stavu q, se vstupním symbolem a, vrcholovým řetězcem zásobníku b, přejde do některého ze stavů pi a vrchol α nahradí příslušným řetězcem $\gamma i \in \Gamma^*$.

Přechod bez čtení vstupního symbolu (e-přechod)
$\delta(q, e, \alpha) = \{ (p1, \gamma1), (p2, \gamma2), \ldots, (pn, \gamma n) \}$

Def.
Rozšířený ZA (RZA) je sedmice $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$
kde δ: $Q \times (\Sigma \cup \{e\}) \times \Gamma^* \rightarrow Q \times \Gamma^*$
tj. reaguje na vrcholové řetězce zásobníku
Př. popsat $\mathcal{P}$ akceptující $L = \{ w\, w^R \}$ kde $w \in \{a, b\}^*$
R má význam „reverzní“
$\mathcal{P} = (\{q,p\}, \{a, b\}, \{a,b,S,Z\}, \delta, q, Z, \{p\})$

cokoliv

$\delta(q,a,e) = \{(q, a)\}$
$\delta(q,b,e) = \{(q, b)\}$
$\delta(q,e,e) = \{(q, S)\}$ to je e-přechod, vloží střed S
$\delta(q,e,aSa) = \{(q, S)\}$ to je e-přechod
$\delta(q,e,bSb) = \{(q, S)\}$ to je e-přechod
$\delta(q,e,ZS) = \{(p, e)\}$

Např akceptace věty abba
$(q, abba, Z) \vdash (q, bba, Za) \vdash (q, ba, Zab) \vdash (q, ba, ZabS) \vdash (q, a, ZabSb) \vdash (q, a, ZaS) \vdash (q, e, ZaSa) \vdash (q, e, ZS) \vdash (q, e, e)$

Def.
Věta w jazyka může být akceptována zásobníkovým automatem $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ dvojím způsobem:

a) přechodem do koncového stavu

$L(\mathcal{P}) = \{ w: (q_0, w, Z_0) \vdash^* (q, e, \gamma), \gamma \in \Gamma^*, q \in F, w \in \Sigma^* \}$

b) s prázdným zásobníkem

$L_e(\mathcal{P}) = \{ w: (q_0, w, Z_0) \vdash^* (q, e, e), q \in Q, w \in \Sigma^* \}$

Vztah bezkontextových gramatik a zásobníkových automatů

Pro danou BKG G = (N, T, P, S) můžeme sestrojit Z A $\mathcal{P}$ takový, že L(G) = L($\mathcal{P}$). Platí i opačně.

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

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

1. **δ(q, e, A) = { (q, α): A → α ∈ P} pro ∀ A ∈ N,**
2. **δ(q, a, a) = { (q, e) } pro ∀a ∈ 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}, δ, q, E, Ø)
δ(q, e, E) = {(q, E+T), (q, T)}
δ(q, e, T) = {(q, T*F), (q, F)}
δ(q, e, F) = {(q, (E)), (q, a)}
δ(q,a',a') = {(q, e)} pro ∀a' ∈ {(,),+,*,a}

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

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

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

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

1. **δ(q, a, e) = { (q, a) } pro ∀a ∈ T,**
2. **δ(q, e, α) = { (q, A) : A → α ∈ P },**
3. **δ(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},δ, q, #, r)**

δ(q, a', e) = { (q, a') } pro ∀a ∈ T,
δ(q, e, E+T) = { (q, E) }
δ(q, e, T) = { (q, E) }
δ(q, e, T*F) = { (q, T) }
…
δ(q, e, #E) = { (r, e) }

Např. zpracování věty a+a

Vrchol

(q, #, a+a) ⊢ (q, #a, +a) ⊢ (q, #F, +a) ⊢ (q, #T, +a)
⊢ (q, #E, +a) ⊢ (q, #E+, a) ⊢ (q, #E+a, e) ⊢ (q, #E+F, e)
⊢ (q, #E+T, e) ⊢ (q, #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.**

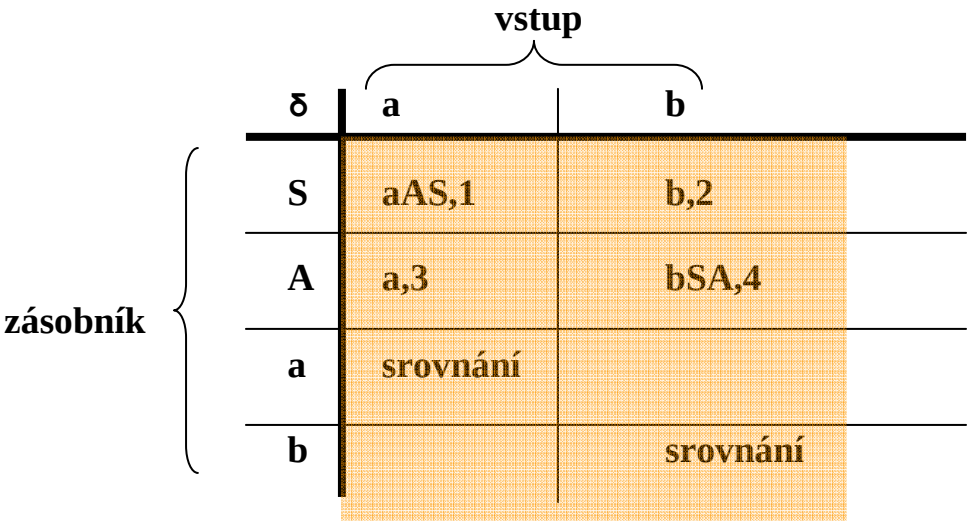
Pozn.: Obsah zásobníku zřetězený se zbytkem vstupu je větnou formou.

LL(k) gramatiky

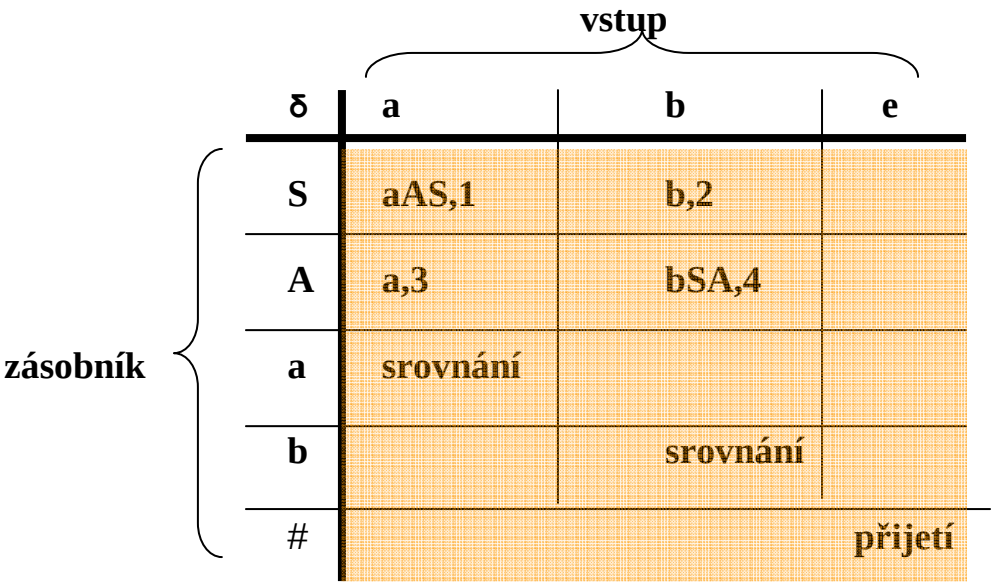
(provádí deterministický rozbor čtením textu z Leva doprava, s použitím Levé derivace a prohlédnutí k dalších symbolů vstupního textu)

Př. $G_1[S]$: **S → a A S (1)**
S → b (2)
A → a (3)
A → bSA (4)

Ekvivalentní ZA **má jediný stav q**



Jak zjistit konec analýzy? (musí být prázdný zásobník i vstup, tzn. v zásobníku bude pouze dno # a prázný vstup signalizuje e)



To je tzv. rozkladová tabulka M, prázdná políčka znamenají chybu

Je to příliš <u>jednoduchá</u> gramatika (simple S-grammer)
Jak lze pro jednoduchou gramatiku vytvořit tabulku M?
Jaké vlastnosti gramatika musí mít, aby M byla jednoznačná?

M: (N ∪ T ∪ {#}) x (T ∪ {e}) → {č.pravidla, srovnání, přijetí}

1. **Jesliže A → a α je i-té pravidlo v P, pak M(A, a) = a α, i**
2. **M(a, a) = srovnání pro všechna a ∈ T**
3. **M(#, e) = přijetí**
4. **M(ostatní) = chyba**

Algoritmus SA pro jednoduché, ..., LL(1) gramatiky

Bude provádět přechody dle 1. nebo 2., dokud nenastane 3. nebo 4.

1. **Je-li M(A, a) = β, i pak (ax, Aα, Π) ├ (ax, βα, Πi)**

2. **Je-li M(a, a) = srovnání pak (ax, aα, Π) ├ (x, α, Π)**

3. V konfiguraci (e, #, Π) končí SA přijetím, Π je levou derivací.

4. Ostatní případy končí chybou.

Konfigurací automatu je trojice (vstup,zásobník,čísla_použitých_pravidel)
!Vrchol zásobníku je vlevo. !

Př. Proveďte v G_1 analýzu věty abbab (na tabuli)
(abbab, S#, -) ├ (abbab, aAS#, 1) ├ (bbab, AS#, 1) ├ (bab, bSAS#, 14) …

Expanze dle pr. 1 srovnání expanze dle pravidla 2

Problém s e-řetězci (co bude na řadě ve vstupu při expanzi A → e ?
To, co se ve větných formách může objevit za A)

Př. G_2[S]: **1** **S → a A S**
2 **S → b**
3 **A → c S A**
4 **A → e**

(tabulku doplnit s výkladem na tabuli)

M =

δ	a	b	c	e
S	aAS, 1	b, 2		
A	e, 4	e, 4	cSA, 3	
a	srovnani			
b		srovnani		
c			srovnani	
#				přijetí

Algoritmus SA je stejný jako u jednoduchých, jen konstrukce M se liší

Např. (na tabuli) **analyzuj v G_2 řetězec** **a b**

(ab, S#, -) ⊢ (ab, aAS#, 1) ⊢ (b, AS#, 1) ⊢ (b, S#, 1 4) ⊢

Expanze dle 1 **srovnání** **expanze dle 4**

⊢ (b, b#, 1 4 2) ⊢ (e, #, 1 4 2)

Expanze dle 2 **srovnání** **výstup = levá derivace věty ab**

Zavedeme funkci **FOLLOW(X) ,** **kde X ∈ N**

FOLLOW (X) =
$\{ c : S =>^* \alpha X \beta, \beta =>^* c \gamma, c \in T, \gamma \in (N \cup T)^* \} \cup \{ e; S =>^* \alpha X \}$

Algoritmus výpočtu FOLLOW(A)

Vstup: Upravená gramatika G, neterminální symbol A

Metoda:

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)**

Př. Co obsahuje FOLLOW(A) v G_2 ?

Použili jsme funkci FIRST (α) kde $\alpha \in (N \cup T)^*$

FIRST (α) = { a; α =>* a β, a ∈ T, β ∈ (N ∪ T)* } ∪ {e; α =>* e }

Algoritmus výpočtu FIRST (α)

Vstup: Upravená gramatika, řetězec α

Metoda:

1. **Je-li α = e, pak FIRST(α) = {e}**
2. **Je-li α = a, pak FIRST(α) = {a}**
3. **Je-li α = X; X ∈ N, X → α_1 | α_2 | … | α_n jsou všechna X pravidla, pak**

$$FIRST(\alpha) = \bigcup_{i=1}^{n} FIRST(\alpha_i)$$

4. **Je-li α = X β ; X ∈ Ne , β ∈ (N ∪ T)***
   pak FIRST(α) = (FIRST(X) - {e}) ∪ FIRST(β)
5. **Jinak α = X β ; X ∈ (N – Ne) ∪ T , pak FIRST(α) = FIRST(X)**

Př. Spočti FIRST a FOLLOW pro

G_3[S]: S → A B A → a A | e B → b B | e

Výpočet hodnot funkcí FIRST a FOLLOW

(alternativa pro programovou realizaci)

Algoritmus

Výpočet funkce FIRST.

Vstup: **Bezkontextová gramatika $G=(N, T, P, S)$ a řetězec $\beta = X_1 X_2 \ldots X_n \in (N \cup T)^*$.**
Výstup: FIRST(β).
Metoda:

1. **Vytvoříme množinu $\mathcal{F}$ takto:**
   (a) $\mathcal{F} = \{ . X_1 X_2 \ldots X_n \}$.
   (b) Jestliže v množině $\mathcal{F}$ je prvek, ve kterém je bezprostředně za tečkou neterminální
   symbol A, přidáme do množiny $\mathcal{F}$ všechna pravidla z P se symbolem A na levé straně a tečku umístíme před první symbol pravé strany:
   $$\mathcal{F} = \mathcal{F} \cup \{ A \to .\alpha : B \to \gamma A \delta \in \mathcal{F}, A \in N, A \to \alpha \in P \}.$$
   (c) Jestliže v množině $\mathcal{F}$ je prvek, ve kterém je tečka na konci pravidla, tj. položka tvaru $B \to \delta.$, vložíme do $\mathcal{F}$ nové položky vytvořené tak, že posuneme tečky za B
   ve všech položkách z $\mathcal{F}$ takových, kde tečka byla před B:
   $$\mathcal{F} = \mathcal{F} \cup \{ A \to \alpha B . \gamma : A \to \alpha . B \gamma \in \mathcal{F}, B \to \delta . \in \mathcal{F} \}.$$
   (d) Kroky b) a c) opakujeme tak dlouho, dokud je možno do $\mathcal{F}$ přidávat další prvky.

2. **Množinu FIRST(β) vytvoříme tak, že do ní vložíme všechny terminální symboly, které se vyskytují bezprostředně za tečkou v některém prvku množiny $\mathcal{F}$. Jestliže v množině $\mathcal{F}$ je prvek, kde se vyskytuje tečka na konci řetězce β, přidáme do FIRST(β) prázdný řetězec:**
   $$\text{FIRST}(\beta) = \{ a : a \in T, A \to \alpha . a \gamma \in \mathcal{F} \} \cup \{ e : \beta . \in \mathcal{F} \}$$

Příklad

Je dána gramatika $G_4 = (\{E,Z,T,D,F\},\{+,*,(,),a\},P,S)$, kde P obsahuje pravidla:

$E \to T\ Z$ $\qquad$ $Z \to +T\ Z \mid e$
$T \to F\ D$ $\qquad$ $D \to *\ F\ D \mid e$
$F \to (E) \mid a$

FIRST(E) vypočteme takto: $\mathcal{F} = \{. E, E \to .T\ Z, T \to .F\ D, F \to .(E), F \to .a\}$, a proto FIRST($E$) = { (, a }.
FIRST(Z) vypočteme takto: $\mathcal{F} = \{.Z, Z \to .+\ T\ Z, Z \to ., Z.\}$, a proto FIRST($Z$) = { + , e }.
FIRST(DZ) vypočteme takto: $\mathcal{F} = \{.DZ, D \to .*F\ D, D \to ., D.Z, Z \to .\ +\ TZ, Z \to ., DZ.\}$ a proto FIRST(DZ) = { *, +, e}.

Na podobném principu jako výpočet funkce FIRST můžeme sestrojit algoritmus pro výpočet funkce FOLLOW.

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$\} \cup \{e : S \rightarrow \alpha . \in F\}$.

Př. Spočti FOLLOW pro symboly z G_4

Algoritmus vytvoření M pro gramatiku s e-pravidly

M je definována na kartézském součinu (N ∪ T ∪ {#}) x (T ∪ {e})

1. Je-li A → a α i-té pravidlo v P, pak M(A, a) = a α, i

2. Je-li A → e i-té pravidlo v P, pak M(A, b) = e, i
pro všechna b ∈ FOLLOW(A)

3. M(a, a) = srovnání pro všechna a ∈ T

3. M(#, e) = přijetí

4. M(ostatní) = chyba

Př. G_5: S → a A
S → b
A → c S a
A → e

?zjistěme FOLLOW (A) = {a, e}
?jak vypadá rozkladová tabulka

	a	b	c	e
S	**aA,1**	**b, 2**		
A	**e, 4**		**cSa, 3**	**e, 4**
a	**sr**			
b		**sr**		
c			**sr**	
#				**přij**

Proveďme rozklad zvolené věty

LL(1) gramatiky

Algoritmus SA zůstává stejný, vytvoření M se liší.

Př. G_6[E]

1 E → T E’
2 E’ → + T E’
3 E’ → e
4 T → F T’
5 T’ → * F T’
6 T’ → e
7 F → (E)
8 F → a

FIRST (T E’) = {a, (}
FIRST (+ T E’) = {+}
FOLLOW (E’) = {e,)}
FIRST (F T’) = {a, (}
FIRST (* F T’) = {*}
FOLLOW (T’) = {e,), +}
FIRST ((E)) = {(}
FIRST (a) = a

M =

	a	+	*	(	)	e
E	TE’, 1			TE’, 1		
E’		+TE’, 2			e, 3	e,3
T	FT’, 4			FT’, 4		
T’		e, 6	*FT’, 5		e, 6	e,6
F	a, 8			(E), 7		
a						
+						
*						
(						
)						
#						

Spodní část M je stále diagonála, zahrneme ji do algoritmu analýzy

Př rozkladu zvolené věty.

Princip syntaktické analýzy 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 $w = a_1\, a_2 \ldots a_n$, který je větou z $L(G)$. Pak existuje levá derivace

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

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

$$\gamma_i = a_1\, a_2 \ldots a_j\, A_i\, \beta_i,$$

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

Vlastnosti algoritmu LL syntaktické analýzy

Předpokládejme, že $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ 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 γ_{i+1}.

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

a) **informaci o dosavadním průběhu (historii) analýzy,**
b) **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:

a) ***Expanze* – neterminální symbol na vrcholu zásobníku je nahrazen pravou stranou vybraného pravidla**
b) ***Srovnání* – terminální symbol na vrcholu zásobníku se ze zásobníku vyloučí, jestliže je shodný se symbolem, který byl ze vstupního řetězce přečten.**
c) ***Přijetí* – vstupní řetězec je přečten a zásobník je prázdný.**
d) ***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.

V případě, že je nutno použít při syntaktické analýze pro rozhodování o dalším postupu informace o historii analýzy, je LL(k) analyzátor řízen rozkladovým automatem a do zásobníku jsou ukládány stavy tohoto automatu.

Z hlediska praktických aplikací jsou nejzajímavější *LL*(1) gramatiky.
Zopakujme- Dána gramatika $G = (N, T, P, S)$, $\alpha \in (N \cup T)^*$, $X \in N$, pak:
FIRST(α) = $\{a : \alpha \Rightarrow^* a\beta, a \in T, \beta \in (N \cup T)^*\} \cup \{e : \alpha \Rightarrow^* e\}$,
FOLLOW(X) = $\{a : S \Rightarrow^* \alpha X \beta, \beta \Rightarrow^* a\gamma, \gamma \in (N \cup T)^*\}$
$\cup \{e : S \Rightarrow^* \alpha X\}$.

Bezkontextová gramatika G=(N, T, P, S) se nazývá *LL*(1) gramatika, když platí následující podmínka pro každé $A \in N$: jestliže $A \to \alpha$ a $A \to \beta$ jsou různá pravidla v P, pak:

a) **Pro řetězce α, β musí platit, že**
   FIRST(α) $\cap$ FIRST(β)=$\varnothing$.
b) **Jestliže z řetězce α lze derivovat prázdný řetězec a z řetězce β nelze derivovat prázdný řetězec, pak musí platit, že**
   FOLLOW(A) $\cap$ FIRST(β)=$\varnothing$.
c) **Jestliže z řetězce α nelze derivovat prázdný řetězec a z β lze derivovat prázdný řetězec, pak musí platit, že**
   FIRST(α) $\cap$ FOLLOW(A)=$\varnothing$.

Poznámka: Předchozí definici je možno zkráceně formulovat takto:
Bezkontextová gramatika G = (N, T, P, S) se nazývá *LL*(1) gramatika, když pro každé $A \in N$ platí podmínka:
Jestliže $A \to \alpha$ a $A \to \beta$ jsou různá pravidla v P, pak
FIRST(α FOLLOW(A)) $\cap$ FIRST(β FOLLOW(A)) = $\varnothing$.

Př. na tabuli ověřit, je-li G_6[E] gramatikou LL(1)

Algoritmus

Vytvoření rozkladové tabulky pro *LL*(1) gramatiku.

Vstup: *LL*(1) gramatika $G = (N, T, P, S)$.
Výstup: Rozkladová tabulka *M* pro gramatiku *G*.
Metoda: Rozkladová tabulka *M* je definována na $N \times (T \cup \{e\})$.

1. **Je-li $A \to \alpha$ i-té pravidlo v *P*, pak $M(A,a) = \alpha, i$ pro všechna $a \in$ FIRST(α) – {*e*}.**
2. **Je-li $A \to \alpha$ i-té pravidlo v *P* a $e \in$ FIRST(α), pak $M(A,b) = \alpha, i$ pro všechna $b \in$ FOLLOW(*A*).**
3. **M(*A*,*a*) = chyba ve všech ostatních případech.**

Algoritmus

Syntaktická analýza *LL*(1) gramatik.

Vstup: Rozkladová tabulka *M* pro *LL*(1) gramatiku $G=(N, T, P, S)$, vstupní řetězec $\omega \in T^*$.
Výstup: Levý rozklad v případě, že $\omega \in L(G)$, jinak chybová signalizace.
Algoritmus čte vstupní řetězec, používá zásobník a vytváří výstupní řetězec. Konfigurace algoritmu je trojice (x, α, π), kde $x \in T^*$ je dosud nepřečtená část vstupního řetězce, $\alpha \in (N \cup T)^*$ je obsah zásobníku a $\pi \in C^*$ je posloupnost čísel pravidel použitých při dosud provedených expanzích. Na začátku je v zásobníku symbol S (zásobník má vrchol vlevo), výstupní řetězec je prázdný.
Metoda: Algoritmus provádí přechody podle 1) a 2), dokud se neobjeví situace podle 3) nebo 4).

1. ***Expanze:* $(ax, A\alpha, \pi) \vdash (ax, \beta\alpha, \pi i)$, jestliže $A \in N$ a $M(A,a) = \beta, i$. Symbol *A* na vrcholu zásobníku je nahrazen řetězcem β a číslo pravidla *i* je připojeno k výstupní posloupnosti.**
2. ***Srovnání:* $(ax, a\alpha, \pi) \vdash (x, \alpha, \pi)$, jestliže $a \in T$. Symbol na vstupu se srovnává se symbolem v zásobníku a v případě rovnosti se vstupní symbol přečte a zásobníkový symbol se ze zásobníku vyloučí.**
3. ***Přijetí:* V konfiguraci $(e, \#, \pi)$ analýza končí a π je levý rozklad vstupního řetězce.**
4. ***Chyba:* Ve všech ostatních případech analýza končí chybovou signalizací.**

Př. na tabuli G_7[E]: $E \to T\ Z$ $Z \to +T\ Z \mid e$ $T \to F\ D$
$D \to *\ F\ D \mid e$ $F \to (\ E\ ) \mid a$

Pro nepřítomné je výsledek na další straně

Výsledek pro G_7[E]:

	a	+	*	(	)	e
E	**TZ,1**			**TZ,1**		
T	**FD,4**			**FD,4**		
Z		**+TZ,2**			**e,3**	**e,3**
D		**e,6**	***FD,5**		**e,6**	**e,6**
F	**a,8**			**(E),7**		

Zkusit analyzovat nějakou větu

(a+a*a, E#, -) ├ (a+a*a, TZ#, 1) ├ (a+a*a, FDZ#, 1 4)
(a+a*a, aDZ#, 1 4 8) ├ (+a*a, DZ#, 1 4 8) ├
├ …

Silné LL(k) gramatiky

(mohou prohlédnout k symbolů ve vstupujícím řetězci)

Def.:

$$FIRST_k(\alpha) = \{ x: \quad x \in T^*, \alpha =>^* x\,y, \; y \in (N \cup T)^*, \; |x| = k\}$$
$$\cup \{ x: \quad x \in T^*, \alpha =>^* x, \; |x| < k\}$$

Def.:

$$FOLLOW_k(A) = \{ x: \; x \in T^*, S =>^* w\,A\,x\,y, \; |x| = k \}$$
$$\cup \{ x: \; x \in T^*, \; S =>^* w\,A\,x, \; |x| < k\}$$

Def.: Gramatika G se nazývá silná LL(k), když pro všechna A ∈ N platí: Jsou-li A → α , A → β různá pravidla v P , pak

$$FIRST_k(\alpha\ FOLLOW_k(A)) \cap FIRST_k(\beta\ FOLLOW_k(A)) = \theta$$

Def. Gramatika je LL(k) pro k ≥ 0, jestliže v případě existence dvou levých derivací

$$S =>^* w\,A\,\alpha => w\,\beta\,\alpha =>^* w\,x$$
$$S =>^* w\,A\,\alpha => w\,\gamma\,\alpha =>^* w\,y$$

takových, že platí $FIRST_k(x) = FIRST_k(y)$
plyne, že β = γ

Věta: G je LL(k) tehdy, platí-li pro ∀ větné formy (no jo, ale kdo je má všechny zjistit) wAα v případě, že

$$A \rightarrow \gamma \;, A \rightarrow \beta$$

jsou dvě různá pravidla, pak

$$FIRST_k(\beta\,\alpha) \cap FIRST_k(\gamma\,\alpha) = \theta.$$

Př. na tabuli zjistit, zda G_8[S]: $S \rightarrow a\,b\,A \mid e$
$A \rightarrow S\,a\,a \mid b$ **je silná LL(k) ?**

Př. Zkusme ještě G[S]: $S \rightarrow a\,A\,a\,a \mid b\,A\,b\,a$
$A \rightarrow b \mid e$

Algoritmus vytvoření rozkladové tabulky pro silné LL(k) gramatiky

1. **Je-li $A \rightarrow \alpha$ i-té pravidlo, v P, pak**
   $M(A, x) = \alpha, i$ pro $\forall x \in FIRST_k(\alpha)$ taková, že $|x| = k$

2. **Je-li $A \rightarrow \alpha$ i-té pravidlo, v P a $y \in FIRST_k(\alpha)$ takové, že $|y| < k$, pak**
   $M(A, z) = \alpha, i$ pro $\forall z \in FIRST_k(y\, FOLLOW_k(A))$

3. **M(Y, x) = chyba v ostatních případech.**

Př. vytvořit M pro $G_8[S]$ na tabuli

	aa	ab	a	ba	bb	b	e
S	e,1	abA,2					e,1
A	Saa,3	Saa,3		b,4		b,4	
a	sr	sr	sr				
b				sr	sr	sr	
#							přj

Algoritmus syntaktické analýzy pro silné LL(k) gramatiky:

Vykonává kroky 1. a 2., dokud nenastane 3. nebo 4.
Označme $u = FIRST_k(x)$

1. **Expanze: $(x, A\alpha, \Pi) \vdash (x, \beta\alpha, \Pi i)$, je-li $M(A, u) = \beta, i$**

2. **Srovnání: $(ax, a\alpha, \Pi) \vdash (x, \alpha, \Pi)$**

3. **Přijetí: V konfiguraci $(e, \#, \Pi)$ končíme přijetím. Π je levou derivací.**
4. **Chyba: Ve všech ostatních případech.**

Př. analýzy věty na tabuli
$(ababbaa, S\#, -) \vdash (ababbaa, abA\#, 2) \vdash^2 (abbaa, A\#, 2) \vdash$
$\vdash \ldots$

LL(1) transformace gramatik

Hledáme k BKG (nelevorekurzivní, bez zbytečných symbolů) ekvivalentní LL(1) gramatiku

Důvody nesplnění LL(1) podmínky mohou být dva:

I. Kolize First First (FF):

Existují alespoň dvě pravidla $A \rightarrow \alpha_1 \mid \alpha_2$, pro která platí

$$FIRST(\alpha_1) \cap FIRST(\alpha_2) \neq \theta$$

II. Kolize First Follow (FFL):

Existují alespoň dvě pravidla $A \rightarrow \alpha_1 \mid \alpha_2$, pro která platí:

$$FIRST(\alpha_1) \cap FOLLOW(A) \neq \theta$$

ad I) Způsobeno výskytem pravidel

$$A \rightarrow \alpha\,\alpha_1 \mid \alpha\,\alpha_2 \mid \ldots \mid \alpha\,\alpha_n$$

Lze někdy odstranit tzv. "levou faktorizací"
Transformujeme G = (N, T, P, S) , kde

$$A \rightarrow \alpha\,\alpha_1 \mid \alpha\,\alpha_2 \mid \ldots \mid \alpha\,\alpha_n \in P_A, \quad \alpha \neq e$$

na
G' = (N', T, P', S), ve které

$$P' = (P - P_A) \cup \{ A \rightarrow \alpha\,A',\ A' \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \}$$

Na faktorizovatelný tvar lze převést eliminací pravidel.
Rezultativnost odstranění FF kolize se nezaručuje

Př. V pravidlech $A \rightarrow a\,B \mid C\,D$ $C \rightarrow a\,E \mid b\,F$ …
odstraňme FF kolizi (se zdůvodněním na tabuli)
Nejprve eliminujeme 2. pravidlo: $A \rightarrow aB \mid aED \mid bFD$
Pak již lze faktorizovat: $A \rightarrow aA_1 \mid bFD$
$A_1 \rightarrow B \mid ED$
$C \rightarrow aE \mid bF$
…

ad II) Způsobeno výskytem dvou pravidel

$B \rightarrow \gamma_1 \mid \gamma_2$ takových, že

$\gamma_1 \Rightarrow^* e$ a zároveň $FIRST(\gamma_2) \cap FOLLOW(B) = T_F \neq \theta$

Lze odstranit tzv. "pohlcením terminálu", který patří do FOLLOW

Předpokládejme, že G obsahuje pravidla:

$B \rightarrow \gamma_2$ pro něž platí $a \in FIRST(\gamma_2)$,

$B \rightarrow \gamma_1$ s vlastností $\gamma_1 \Rightarrow^* e$ a také pravidlo
$A \rightarrow \alpha B a \beta$

Transformujeme původní $G = (N, T, P, S)$, kde

$A \rightarrow \alpha B a \beta \in P$

$B \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \in P$ jsou všechna B pravidla

na novou gramatiku $G' = (N', T, P', S)$, ve které

$N' = N \cup [Ba]$, kde $[Ba]$ je nový neterminál

$P' = P - \{A \rightarrow \alpha B a \beta\} \cup \{A \rightarrow \alpha [B a] \beta\} \cup$
$\cup \{[B a] \rightarrow \alpha_1 a \mid \alpha_2 a \mid \ldots \mid \alpha_n a\}$

Odstraněním FF kolize můžeme ale způsobit FFL a naopak.
Rezultativnost LL(1) transformace se nezaručuje.

Př. V pravidlech $A \rightarrow B a C$ $B \rightarrow e \mid a b C$ $C \rightarrow e \mid c B C$
odstraňte FFL kolizi
Zavedeme neterminál [Ba]
$[Ba] \rightarrow a \mid a b C a$
A nahradíme jím část pravé strany 1.pravidla.
$A \rightarrow [Ba] C$
$B \rightarrow e \mid a b C$ zde již kolize FFL není
$C \rightarrow e \mid c B C$
$[Ba] \rightarrow a \mid a b C a$ tady ale vznikla FF

Př. Odstraňte kolize vzniklé v G[E] po eliminaci levé rekurze. Na tabuli:
$E \rightarrow T \mid T E'$
$E' \rightarrow +T \mid + T E'$
$T \rightarrow F \mid F T'$
$T' \rightarrow * F \mid * F T'$
$F \rightarrow (E) \mid a$

Řešení pro pohodlné

Provedeme faktorizaci:

$E \rightarrow T\ E_1$
$E_1 \rightarrow E' \mid e$
$E' \rightarrow +\ T\ E_2$
$E_2 \rightarrow E' \mid e$
$T \rightarrow F\ T_1$
$T_1 \rightarrow T' \mid e$
$T' \rightarrow *\ F\ T_2$
$T_2 \rightarrow T' \mid e$
$F \rightarrow (\ E\ ) \mid a$

E_1 a E_2 mají tutéž generační schopnost, jedno z nich je proto zbytečné a můžeme je nahradit druhým.
T_1 a T_2 mají tutéž generační schopnost, jedno z nich je proto zbytečné a můžeme je nahradit druhým.

$E \rightarrow T\ E_1$
$E_1 \rightarrow E' \mid e$
$E' \rightarrow +\ T\ E_1$
$T \rightarrow F\ T_1$
$T_1 \rightarrow T' \mid e$
$T' \rightarrow *\ F\ T_1$
$F \rightarrow (\ E\ ) \mid a$

Vyloučíme E' a T' dosazením jejich pravých stran a dostaneme

$E \rightarrow T\ E_1$
$E_1 \rightarrow +\ T\ E_1 \mid e$
$T \rightarrow F\ T_1$
$T_1 \rightarrow *\ F\ T_1 \mid e$
$F \rightarrow (\ E\ ) \mid a$

Což je výsledek, který při použití kratší verze odstraňování levé rekurze bychom dostali rovnou.

LR(k) gramatiky

Př. Uvažujme gramatiku $G_{10}[S]$

1. $S \rightarrow A\ B$
2. $A \rightarrow f$
3. $A \rightarrow a$
4. $B \rightarrow b\ C$
5. $C \rightarrow c$

Rozšířený ZA bude mít přechod. fci (viz body z předn. 8 BKG:

dle 1) $\delta(q, i, -) = \{(q, i)\}$ pro $\forall i \in \{a, b, c, f\}$

dle 2) $\delta(q, -, A\ B) = \{(q, S)\}$
$\delta(q, -, f) = \{(q, A)\}$
$\delta(q, -, a) = \{(q, A)\}$
$\delta(q, -, b\ C) = \{(q, B)\}$
$\delta(q, -, c) = \{(q, C)\}$

dle 3) $\delta(q, e, \# S) = \{(r, e)\}$

Takový automat je nedeterministický, ! vrchol zásobníku je vpravo !

δ provádí operace: přesouvání , dle 1)
redukování, dle 2)
akceptování. dle 3)

Jeho konfigurací je trojice (stav, vstup, obsah zásobníku)

např. zpracování řetězce f b c (na tabuli)

$(q, fbc, \#) \vdash (q, bc, \#f) \vdash (q, bc, \#A) \vdash (q, c, \#Ab) \vdash \ldots$

Přesun redukce dle 2 přesun

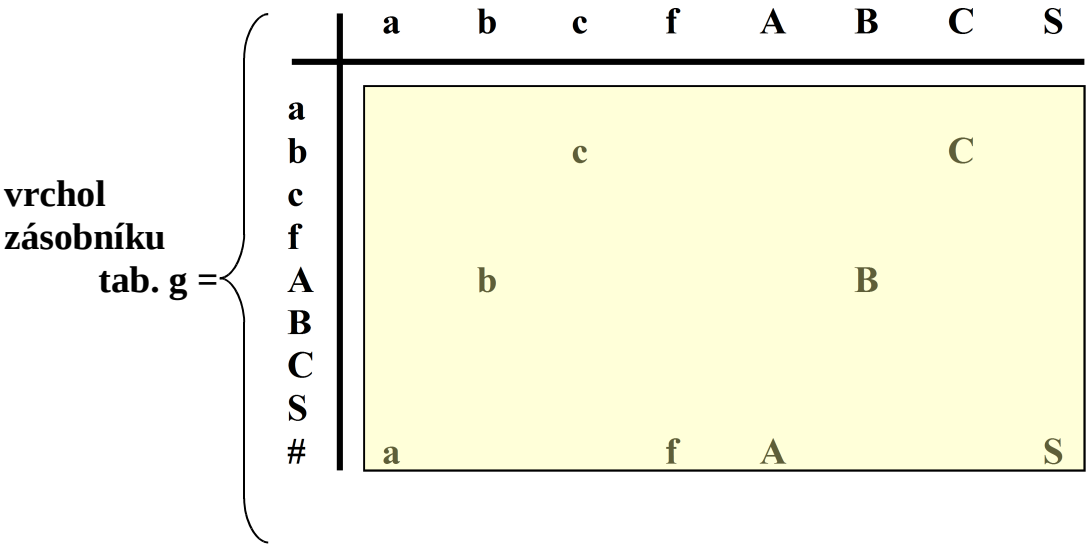
δ je nepřehledné, použijeme tabulky:

tabulka akcí f (co má ZA udělat za operaci)
tabulka přechodů g (co má vložit do zásobníku)

tab.f =

vrchol zásobníku	akce
a	redukce 3
b	přesun
c	redukce 5
f	redukce 2
A	přesun
B	redukce 1
C	redukce 4
S	přijetí
#	přesun

Vkládaný symbol do zásobníku



Tak snadné to je jen u tzv.<u>triviálních</u> gramatik (jaké musí mít vlastnosti?)
Mohou mít rekurzivní pravidla?
Mohou vůbec obsahovat rekurzivní symboly?

Vytvoření tabulky akcí a tabulky přechodů pro triviální LR gramatiku

1. **Tabulka akcí f, (řádky jsou z $N \cup T \cup \{ \# \}$)**

a) **Je-li symbol $X \in N \cup T$ na konci pravidla i: $A \rightarrow \alpha$, pak f(X) = redukce(i),**
b) f(S) = přijetí
c) f(X) = přesun v ostatních případech

2. Tabulka přechodů g (sloupce jsou $N \cup T$, řádky jsou $N \cup T \cup \{ \# \}$)

a) **g(#, X) = X jestliže v G $\exists$ derivace $S \Rightarrow^* X \alpha$**
b) **g(X, Y) = Y jestliže G obsahuje pravidlo $A \rightarrow \alpha X Y \beta$**
c) **g(X, Y) = Y jestliže G obsahuje pravidlo $A \rightarrow \alpha X B \beta$, kde $B \in N$ a $B \Rightarrow^+ Y \gamma$**
d) g(X, Y) = chyba v ostatních případech

Algoritmus SA triviální LR gramatiky (platí i pro LR(0))

Označme symbol na vrcholu zásobníku X, dno označme #.

1.

a. **Je-li f(X) = přesun, přečti vstupní symbol a jdi na 2.**
b. **Je-li f(X) = redukce(i), vyloučí se ze zásobníku pravá strana pravidla i, číslo i se přidá do výstupu a přejde se na bod 2.**
c. **Je-li f(X) = přijetí, pak při zároveň prázdném vstupním řetězci ukončíme činnost akceptací, při neprázdném ukončíme odmítnutím.**

2.

Je-li Y symbol , který má být vložen do zásobníku, provedeme:

a. **Je-li g(X, Y) = Z, uložíme Z na vrchol zásobníku a opakujeme 1.**
b. **Je-li g(X, Y) = chyba, ukončíme analýzu chybou.**

Konfiguraci zapisujme ve tvaru
(obsah zásob. s vrcholem vpravo, zbytek vst. řetězce, čísla pr. pro redukce)

Je to názornější = Tvoří větnou formu

Př. na tabuli analýza řetězce dle tabulek pro $G_{10}[S]$

(#, fbc, -) ├ (#f, bc, -)
├ (#A, bc, 2)
├ (#Ab, c, 2)
├ (#Abc, e, 2)
├ (#AbC, e, 25)
├ (#AB, e, 254)
├ (#S, e, 2541)

LR(0) gramatiky

Při násobném výskytu některého symbolu na pravé straně pravidel, je nutné rozlišovat (třeba indexem) tyto výskyty i v procesu SA, tedy i v zásobníku. Pro nekomplikované G to zvládneme jako u triviálních G.

Př. G_{11}[S]	**1**	**S → B**	**S → B1**
	2	**B → a B b**	**B → a B2 b1**
	3	**B → A**	**B → A1**
	4	**A → b A**	**A → b2 A2**
	5	**A → c**	**A → c**

Sestrojíme na tabuli f a g (pro G_{11} to ještě zvládneme algoritmem pro triviální gramatiku)

Zás	akce	a	b	c	B	A	S
#	přesun	a	b2	c	B1	A1	S
a	přesun	a	b2	c	B2	A1	
b1	R2						
b2	přesun		b2	c		A2	
c	R5						
B1	R1						
B2	přesun		b1				
A1	R3						
A2	R4						
S	akcept						

Př. syntaktické analýzy na tabuli

(#, abcb, -) ⊢ (#a, bcb, -)
⊢ (#ab_2 , cb, -)
⊢ (#ab_2 c, b, -)
⊢ (#ab_2A_2, b, 5)
⊢ (#aA_1, b, 5 4)
⊢ (#aB_2, b, 5 4 3)
⊢ (#a B_2b_1, e, 5 4 3)
⊢ (# B_1, e, 5 4 3 2)
⊢ (#S, e, 5 4 3 2 1)

U komplikovanějších gramatik použijeme místo indexování symbolů k výpočtu tabulek tzv. množiny položek.

Výpočet rozkladových LR tabulek pomocí množin položek

Platí, že g(#, X) = X jestliže v G ∃ derivace S =>* X α

Podle dosavadního postupu

proto

g(#, S) =	**S**
g(#, B) =	**B1**
g(#, a) =	**a**
g(#, A) =	**A1**
g(#, c) =	**c**
g(#, b) =	**b2**

symboly, které mohou být v zásobníku přímo u #

f(#) = přesun

Musíme zjistit jaké situace v konfiguracích mohou při SA nastávat (co lze kdy vkládat do zásobníku) a jaká akce je ta jediná správná.

Situace charakteristická pro určitý vrcholový symbol zásobníku je popsatelná tzv. množinou LR(0) položek.
Pro # na vrcholu to je

vrchol zásobníku **ještě venku = nezpracováno**

 # : S → . B
B → . a B b
B → . A
A → . b A
A → . c

Tečka symbolizuje rozhranní *zásobník . dosud nezpracovaná část vstupu*

Vkládání symbolů do zásobníku (přesouváním ze vstupu nebo redukcemi vrcholového řetězce) je symbolizováno posouváním tečky.

Z množiny pro # plyne, že při vrcholu # mohu k němu vložit B (ale v jaké variantě?) nebo a nebo A (ale v jaké variantě?) nebo b (ale v jaké variantě?) nebo c.

Posouváním tečky dostáváme postupně konečný soubor množin položek a současně i graf přechodů mezi množinami. Tato informace plně popisuje možné stavy při SA.

Algoritmus

Vytvoření souboru množin *LR*(0) položek.

Vstup: Bezkontextová gramatika $G = (N, T, P, S)$.

Výstup: Soubor φ množin *LR*(0) položek pro *G*.

Metoda:

1. **Počáteční množinu *LR*(0) položek M_0 vytvoříme takto:**
   (a) $M_0 = \{S \to \cdot\, \alpha : S \to \alpha \in P\}$.
   (b) Jestliže $A \to \cdot\, B\alpha \in M_0$, $B \in N$ a $B \to \beta \in P$, pak
   $M_0 = M_0 \cup \{B \to \cdot\, \beta\}$.
   (c) Opakujme krok b) tak dlouho, dokud je možné přidávat nové položky do M_0.
   (d) $\varphi = \{M_0\}$, M_0 je počáteční množina.
2. **Jestliže jsme zkonstruovali množinu *LR*(0) položek M_i, zkonstruujeme pro každý symbol $X \in N \cup T$ takový, že leží v některé *LR*(0) položce v M_i za tečkou další množinu *LR*(0) položek M_j, kde *j* je index vetší než nejvyšší index dosud vytvořené množiny M_i takto:**

   (a) $M_j = \{A \to \alpha X \cdot \beta : A \to \alpha \cdot X \beta \in M_i\}$.
   (b) Jestliže $A \to \alpha \cdot B \beta \in M_j$, $B \in N$, $B \to \gamma \in P$, pak
   $M_j = M_j \cup \{B \to \cdot\, \gamma\}$.
   (c) Opakujeme krok (b) tak dlouho, dokud je možné do M_j přidávat nové položky.
   (d) $\varphi = \varphi \cup \{M_j\}$.
3. **Krok 2) opakujeme pro všechny vytvořené množiny M_j, dokud je možné do φ přidávat nové množiny M_j.**

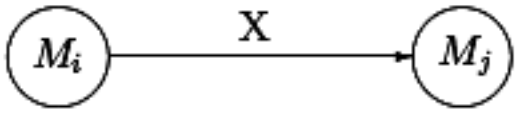
Poznámka: Krok 1a) a 2a) budeme nazývat vytvoření základů množin *LR*(0) položek opakované provádění kroku 1b) a 2b) nazveme vytváření uzávěrů množin položek.

Definice

GOTO (M_i, X) = M_j, jestliže položky tvaru $A \to \alpha \cdot X \beta$ leží v M_i a základ množiny M_j byl vytvořen položkami tvaru $A \to \alpha X \cdot \beta$.

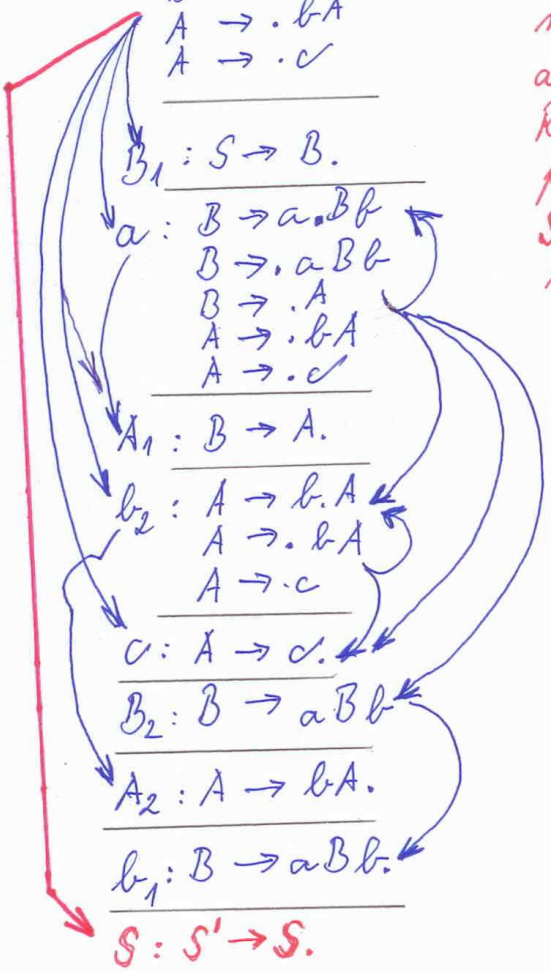
Funkce GOTO si můžeme znázornit jako orientovaný hranově a uzlově ohodnocený graf takto:

Funkci GOTO (M_i, X) = M_j bude odpovídat graf:



Př. na tabuli (zabere téměř stránku)

G: 1) $S \to B$
2) $B \to aBb$
3) $B \to A$
4) $A \to bA$
5) $A \to c$



Když vyplníme podle této modré množiny množin položek tabulky f a g, bude nám chybět údaj do řádku a sloupce pro S. Rozšíříme proto gramatiku o pravidlo 0) $S' \to S$, kde S' bude nový počáteční symbol a doplníme graf

Algoritmus

Konstrukce tabulky akcí f a tabulky přechodů g pro LR(0) gramatiku
Vstup: Soubor LR(0) položek
Výstup: Tabulky f, g

Postup:

1) Řádky f budou odpovídat množinám položek. Sestrojí se následovně:

Je-li v množině položek M obsažena položka tvaru $A \rightarrow \alpha .$
pak f(M) = redukce(i), kde i je číslo pravidla $A \rightarrow \alpha$

Je-li v množině položek X obsažena položka tvaru $S' \rightarrow S.$
pak f(M) = přijetí

f(M) = přesun v ostatních případech

2) **Tabulka přechodů g odpovídá přechodové funkci GOTO mezi množinami položek. Řádky g budou odpovídat množinám položek. Sestrojí se následovně:**

a) **Je-li $GOTO(M_i, X) = M_j$, pak $g(M_i, X) = M_j$**

b) **Je-li $GOTO(M_i, X)$ = prázdná množina, pak $g(M_i, X)$ = chyba**

Př. Konstruujme f a g na základě LR(0) položek. Všimněte si, něco v nich chybí. To je důvod použití rozšířené gramatiky (viz doplnění předchozího grafu přechodů o červenou část)

Zás	akce	a	b	c	B	A	S
#	přesun	a	b2	c	B1	A1	S
a	přesun	a	b2	c	B2	A1	
b1	R2						
b2	přesun		b2	c		A2	
c	R5						
B1	R1						
B2	přesun		b1				
A1	R3						
A2	R4						
S	akcept						

SLR(k) gramatiky

Obsahuje-li některá z množin LR(0) položek

jak položku $A \rightarrow \alpha .$
tak položku $B \rightarrow \beta .$ } **(tzv. konflikt redukce redukce)**
či položku $C \rightarrow \gamma . \delta$ **(tzv. konflikt redukce přesun)**

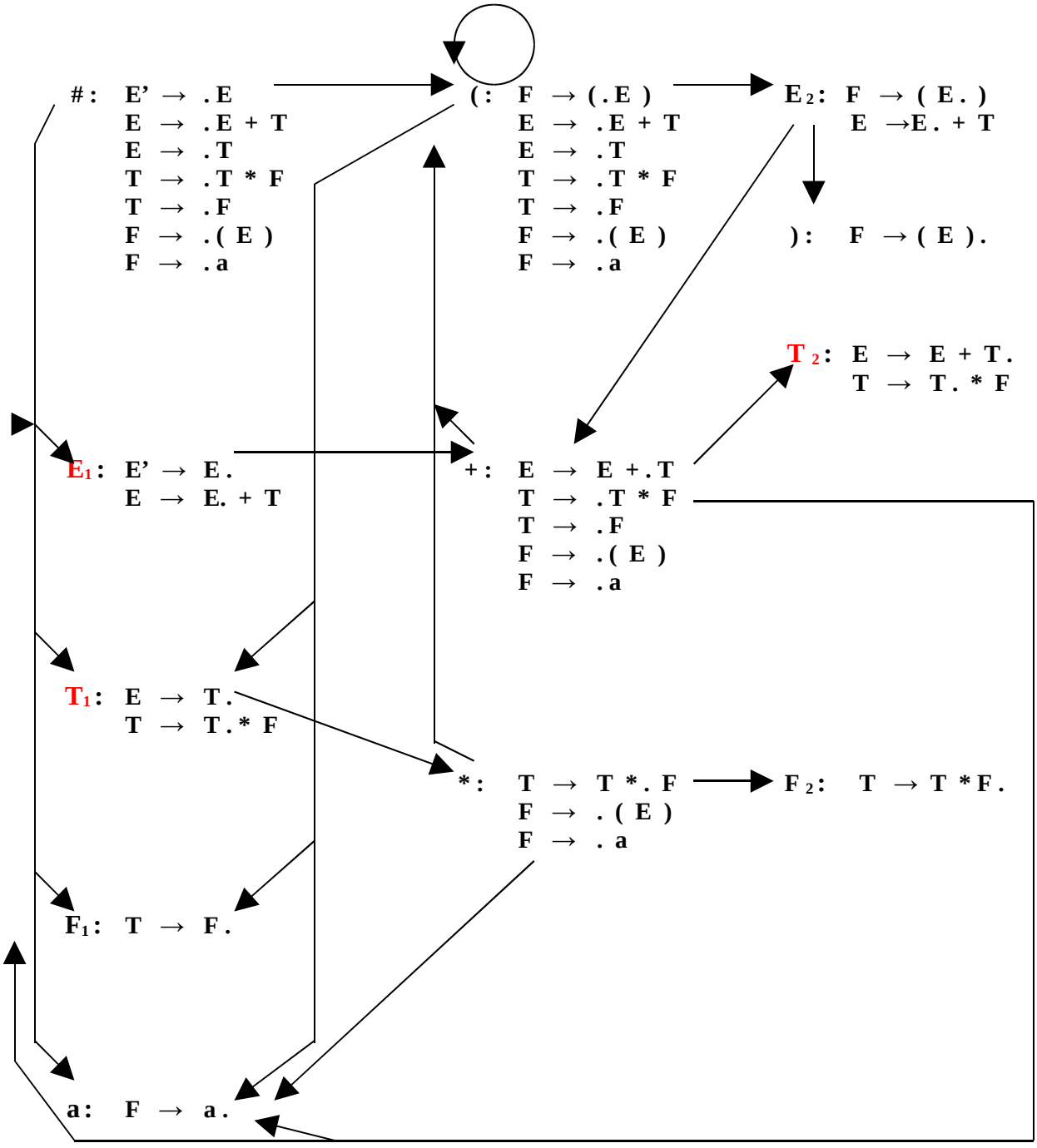
pak gramatika není LR(0). Pokud to půjde napravit prohlížením vstupujících symbolů, pak se jedná o některou z podtříd LR(k) gramatik.

Př. G_{12}**[E']**

0	**E'**	**→**	**E**
1	**E**	**→**	**E + T**
2	**E**	**→**	**T**
3	**T**	**→**	**T * F**
4	**T**	**→**	**F**
5	**F**	**→**	**(E)**
6	**F**	**→**	**a**

Vytvoříme množiny LR(0) položek (na tabuli, téměř stránka)

LR(0) množiny položek pro G[E]



V červeně označených množinách položek nastal konflikt redukce přesun

Předpokládejme, že $A \to \alpha . \beta$ a $B \to \gamma . \delta$ jsou dvě různé položky z M. Jestliže každé dvě takové položky splňují alespoň jednu z podmínek:

1. **Ani β, ani δ nejsou prázdné řetězy,**
2. **$\beta \neq e$, $\delta = e$ a $\text{FOLLOW}_k(B) \cap \text{FIRSTk}(\beta \text{FOLLOW}_k(A)) = \emptyset$ pro $\beta \in T(N \cup T)^*$,**
3. **$\beta = e$, $\delta \neq e$ a $\text{FOLLOW}_k(A) \cap \text{FIRST}_k(\delta \text{FOLLOW}_k(B)) = \emptyset$ pro $\delta \in T(N \cup T)^*$**
4. **$\beta = \delta = e$ a $\text{FOLLOW}_k(A) \cap \text{FOLLOW}_k(B) = \emptyset$,**

pak G se nazývá jednoduchá $LR(k)$ gramatika (zkráceně $SLR(k)$ gramatika). Zápis $\beta \in T(N \cup T)^*$ znamená, že řetěz β začíná terminálním symbolem.

Pro $SLR(k)$ gramatiky můžeme sestrojit tabulku akcí f a tabulku přechodů g pomocí následujícího algoritmu.

Algoritmus

Konstrukce tabulky akcí f a tabulky přechodů g pro $SLR(k)$ gramatiku

Vstup: $SLR(k)$ gramatika $G = (N, T, P, S)$ a soubor množin $LR(0)$ položek φ pro G.

Výstup: Tabulka akcí f a tabulka přechodů g pro G.

Metoda:

1. **Tabulka akcí f bude mít řádky označeny stejně jako množiny položek z φ. Sloupce tabulky budou označeny řetězy symbolů z T^{*k},**
   (a) $f(M_i, u)$ = přesun, jestliže $A \to \beta_1 \cdot \beta_2 \in M_i$, $\beta_2 \in T(N \cup T)^*$ a $u \in \text{FIRST}_k(\beta_2 \text{FOLLOW}_k(A))$,
   (b) $f(M_i, u)$ = redukce(j), jestliže $j \geq 1$ a $A \to \beta \cdot \in M_i$, $A \to \beta$ je j-té pravidlo v P a $u \in \text{FOLLOW}_k(A)$,
   (c) $f(M_i, e)$ = přijetí, jestliže $S' \to S \cdot \in M_i$,
   (d) $f(M_i, u)$ = chyba ve všech ostatních případech.

2. **Tabulka přechodů g odpovídá funkci GOTO.**
   (a) Je-li $\text{GOTO}(M_i, x) = M_j$, kde $x \in (N \cup T)$, pak $g(M_i, x) = M_j$.
   (b) Je-li $\text{GOTO}(M_i, x)$ prázdná množina pro $x \in (N \cup T)$, pak $g(M_i, x)$ = chyba.

Př. na tabuli cca 15 řádek

	a	+	*	(	)	e	E	T	F	a	+	*	(	)
#		P			P			E1	T1	F1	a			(
E1			P				A					+		
		T1		2	P		2	2						*
		F1		4	4		4	4						
		a		6	6		6	6						
		(	P			P			E2	T1	F1	a		(
		+	P			P				T2	F1	a		(
		*	P			P					F2	a		(
		E2		P		P							+	
		)												
		T2		1	P		1	1						*
		F2		3	3		3	3						
		)		5	5		5	5						

Algoritmus

Syntaktická analýza pro *SLR*(*k*) gramatiky (algoritmus je použitelný i pro *LALR*(*k*) gramatiky a pro *LR*(*k*) gramatiky – viz. dále)

Vstup: **Tabulka akcí *f* a tabulka přechodů *g* pro *G* = (*N*, *T*, *P*, *S*), vstupní řetěz $w \in T^*$ a počáteční symbol zásobníku M_0 (označení počáteční množiny *LR*(*0*) položek).**

Výstup: **Pravý rozklad v případě, že $w \in L(G)$, jinak chybová indikace.**

Metoda: **Algoritmus čte symboly ze vstupního řetězu *w*, využívá zásobník a vytváří řetěz čísel pravidel, která byla použita při redukcích. V zásobníku je na začátku symbol M_0.**

Opakujeme kroky 1), 2), a 3) dokud nenastane *přijat* nebo *chyba*.

Symbol X je symbolem na vrcholu zásobníku.

1. **Určíme řetěz *k* prvních symbolů z dosud nepřečtené části vstupního řetězu a označíme jej *u*.**
2. **(a) Je-li *f*(*X*, *u*) = *přesun*, přečte se vstupní symbol a přejdeme na krok 3).**
   (b) Je-li *f*(*X*, *u*) = *redukce*(*i*), vyloučíme ze zásobníku tolik symbolů, kolik je symbolů na pravé straně i-tého pravidla (*i*)$A \rightarrow \alpha$ a do výstupního řetězu připojíme číslo pravidla (*i*). Přejdeme na krok 3).
   (c) Je-li *f*(*X*, *e*) = *přijetí*, ukončíme analýzu a výstupní řetěz je pravý rozklad vstupní věty *w* v případě, že vstupní řetěz je celý přečten, jinak chyba.
   (d) Je-li *f*(*X*, *u*) = *chyba*, ukončíme rozklad chybovou indikací.
3. **Je-li *Y* symbol, který má být uložen do zásobníku (přečtený symbol ve 2a) nebo levá strana pravidla použitého při redukci (v 2b)) a X je symbol na vrcholu zásobníku, pak:**
   (a) Je-li *g*(*X*, *Y*) = *Z*, pak uložíme *Z* na vrcholu zásobníku a opakujeme od kroku 1.
   (b) Je-li *g*(*X*, *Y*) = *chyba*, ukončíme analýzu chybovou indikací.

Konfigurací algoritmu budeme rozumět trojici:

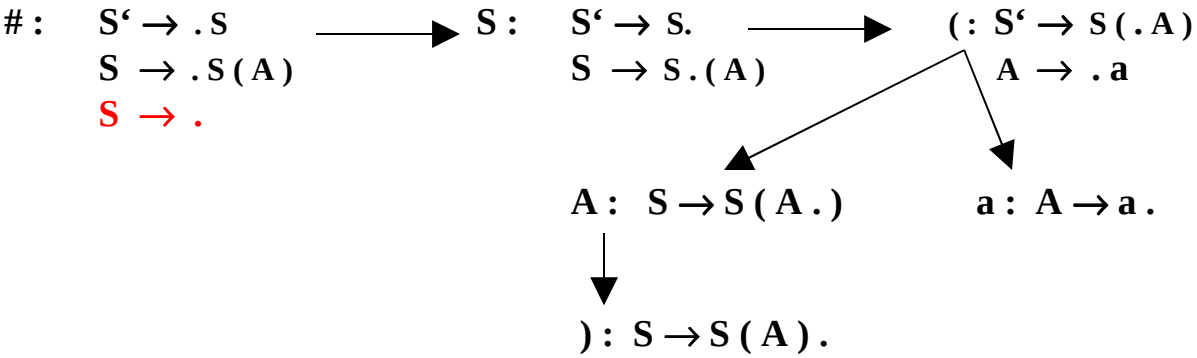
(α, *x*, π), kde α **je obsah zásobníku**
x **je dosud nepřečtená část vstupního řetězce textu,**
π **je dosud vytvořená část výstupního řetězce pravidel.**

Konfigurace (M_0 , *w*, *e*) je počáteční a konfigurace (M_0M_i, *e*, π) je koncová konfigurace algoritmu, kde M_i je symbol na vrcholu zásobníku, při kterém došlo k přijetí (*f*(M_i, *e*)=*přijetí*).

Př. na tabuli pro G_{12}[E']

(#, a+a*a, -) ├ (#a, +a*a, -) ├ (#F_1, +a*a, 6) ├
├ (#T_1, +a*a, 6 4) ├ (#E_1, +a*a, 6 4 2) ├ …

Př. G_{13}[S']: S'→ S S → S (A) | e A → a
S očíslováním (0) (1) (2) (3)
Množiny položek a graf přechodů mezi nimi mají tvar:



Jak to bude s tím e?

Tabulky mají tvar:

	a	(	)	e		S	A	a	(	)
#		2		2		S				
S		P		A					(	
A			P							)
a			3							
(	P						A	a		
)		1		1						

Proč? Protože jak již víme platí:

***f*(#, *u*) = přesun, jestliže $X \to \beta_1 \cdot \beta_2 \in \#$, $\beta_2 \in T(N \cup T)^*$ a**
$u \in$ FIRST (β_2 FOLLOW (X)),
a tento případ zde není

***f*(#, *u*) = redukce(*j*), jestliže $X \to \beta \cdot \in \#$, $X \to \beta$ je j-té pravidlo v *P* a**
$u \in$ FOLLOW$_k$(X)
a tento případ zde je

Zkusme nějakou větu analyzovat
(# , (a) , -) ├ (# S , (a) , 2) ├ (# S (, a) , 2) ├ (# S (a ,) , 2) ├
(# S (A ,) , 2 3) ├ (# S (A) , e , 2 3) ├ (# S , e , 2 3 1)

LALR(k) gramatiky

V množinách LR(0) položek může nastat konflikt redukce-redukce nebo redukce-přesun, který lze odstranit zohledněním dopředu prohlížených symbolů v obecnějších tzv. LALR(k) položkách („look ahead LR(k)“)

Definice

***LR(k)* položka pro bezkontextovou gramatiku**
***G* = (*N*, *T*, *P*, *S*)**
je objekt tvaru:

$[A \rightarrow \alpha \cdot \beta,\ w]$

to, co je za β, nebo-li
co může být právě za tímto A, nebo-li
w∈ podmnožiny FOLLOW(A)

kde A → α β ∈ P a *w* ∈ *T, |*w*| ≤ *k* je tzv. dopředu prohlížený řetěz terminálních symbolů délky nejvýše *k*.**
Dále nadefinujeme pojem jádra položek v množině *LR*(*k*) položek.

Definice
Nechť *M* je množina *LR*(*k*) položek. Jádro *J* množiny položek *M* je množina:

$$J(M) = \{A \rightarrow \alpha \cdot \beta : [A \rightarrow \alpha \cdot \beta, \mathrm{w}] \in M\}.$$

Nyní můžeme uvést algoritmus pro výpočet souboru množin položek pro *LALR*(*k*) gramatiku, tj. ; soubor množin *LALR*(*k*) položek.

Algoritmus

Výpočet souboru množin *LALR*(*k*) položek pro *G* = (*N*, *T*, *P*, *S*)

Vstup: **Bezkontextová gramatika *G* = (*N*, *T*, *P*, *S*).**
Výstup: **Soubor φ množin *LALR*(*k*) položek pro *G*.**
Metoda:

1. **Počáteční množinu *LALR*(*k*) položek M_0 vytvoříme takto:**
   (a) $M_0 = \{[S \rightarrow \cdot\, \omega, e] : S \rightarrow \omega \in P\}$.
   (b) Jestliže $[A \rightarrow \cdot\, B\alpha, u] \in M_0$, $B \in N$ **a** $B \rightarrow \beta \in P$, **pak**
   $M_0 = M_0 \cup \{[B \rightarrow \cdot\, \beta, x]$ **: pro všechna** $x \in \text{FIRST}_k(\alpha u)\}$.
   (c) Opakujeme krok (b) tak dlouho, dokud je možno do M_0 přidávat nové položky.
   (d) $\varphi = \{M_0\}$, M_0 **je počáteční množina.**
2. **Jestliže jsme zkonstruovali množinu *LALR*(*k*) položek M_i, zkonstruujeme pro každý symbol** $X \in (N \cup T)$ **takový, že leží v některé *LALR*(*k*) položce v M_i za tečkou další množinu *LALR*(*k*) položek M_j , kde *j* je vetší než nejvyšší index dosud vytvořené množiny *LALR*(*k*) položek v φ, takto:**
   (a) $M_j = \{[A \rightarrow \alpha X \cdot\, \beta, u] : [A \rightarrow \alpha \cdot\, X\, \beta, u] \in M_i\}$.
   (b) Jestliže $[A \rightarrow \alpha \cdot\, B\, \beta, u] \in M_j$, $B \in N$, $B \rightarrow \gamma \in P$, **pak**
   $M_j = M_j \cup \{[B \rightarrow \cdot\, \gamma, x]$ **: pro všechna** $x \in \text{FIRST}_k(\beta u)\}$.
   (c) Opakujeme krok (b) tak dlouho, dokud je možné do M_j přidávat nové položky.
   (d) Jestliže jádro $J(M_j) \neq J(M_n)$ **pro všechna** $M_n \in \varphi$, **pak** $\varphi = \varphi \cup \{M_j\}$ **a** $\text{GOTO}(M_i, X) = M_j$.
   Jestliže $J(M_j) = J(M_n)$ **pro nějaké** M_n **z** φ, **pak**
   $M_n' = Mn \cup M_j$ **a** $\varphi = (\varphi - \{M_n\}) \cup \{M_n'\}$ **a** $\text{GOTO}(M_i, X) = M_n'$.
3. **Krok 2. opakujeme pro všechny vytvořené množiny v** φ **tak dlouho, dokud je možné vytvářet nové množiny M_j .**

Definice

Bezkontextovou gramatiku *G* = (*N*, *T*, *P*, *S*) nazveme *LALR*(*k*) gramatikou (*k* ≥ 0) právě tehdy, když v souboru množin *LALR*(*k*) položek vytvořených podle algoritmu pro rozšířenou gramatiku *G'* nejsou žádné konflikty.

Poznámka:

V dalších případech budeme pro položky

$[A \to \alpha \cdot \beta, x_1]$, $[A \to \alpha \cdot \beta, x_2]$, … $[A \to \alpha \cdot \beta, x_n]$

používat zkráceného zápisu $[A \to \alpha \cdot \beta, x_1|x_2| \ldots |x_n]$.

Jestliže v množině *LALR*(*k*) položek nejsou žádné konflikty, můžeme sestrojit tabulku akcí *f* pro syntaktický analyzátor pomocí následujícího algoritmu.
Tabulka přechodů se vytvoří stejně jako pro *SLR*(*k*) gramatiku na základě funkce GOTO.

Algoritmus sestrojení tabulky akcí pro *LALR*(*k*) gramatiku.

Vstup: Soubor množin φ *LALR*(*k*) položek pro gramatiku $G = (N, T, P, S)$.
Výstup: Tabulka akcí *f* pro gramatiku *G*.

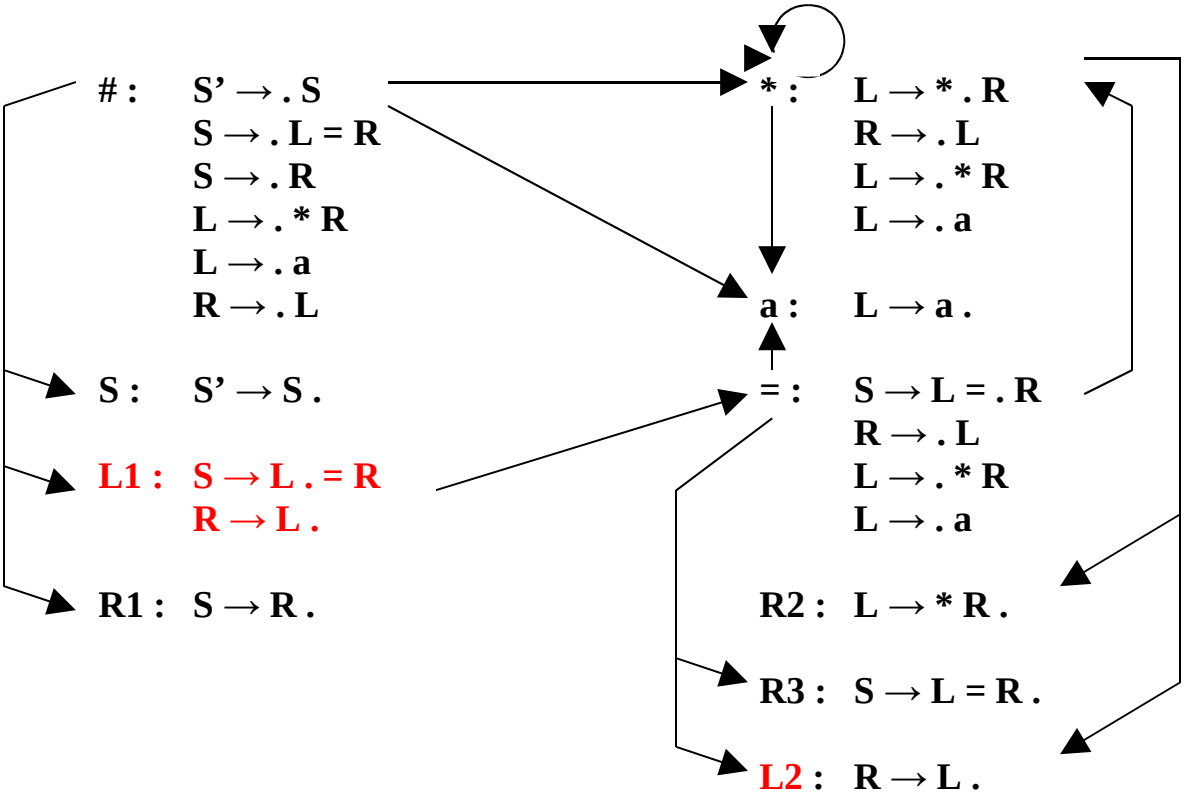
Metoda: Tabulka akcí *f* bude mít řádky označeny stejně jako množiny z φ.
Sloupce budou označeny řetězy symbolů $u \in T^*$, $|u| \leq k$.
Pro všechna $i \in \langle 0, |\varphi| \rangle$ provedeme:

a) $f(M_i, u)$ **= přesun, jestliže** $[A \to \beta_1 \cdot \beta_2, v] \in M_i$, $\beta_2 \in T(N \cup T)^*$ **a** $u \in$ **FIRSTk(**β_2 v**).**
b) $f(M_i, u)$ **= redukce(*j*), jestliže** $[A \to \beta \cdot, u] \in M_i$ **a** $A \to \beta$ **je j-té pravidlo v *P*, kromě situace podle c),**
c) $f(M_i, e)$ **= přijetí, jestliže** $[S' \to S \cdot, e] \in M_i$ **a vstupní řetěz je přečten,**
d) $f(M_i, n)$ **= chyba v ostatních případech.**

Př. G_{14} **na tabuli ? (stránka)**

$S \to L = R$
$S \to R$
$L \to * R$
$L \to a$
$R \to L$

Zkonstruujme soubor množin LR(0) položek pro rozšířenou gramatiku $G_{14}[S']$

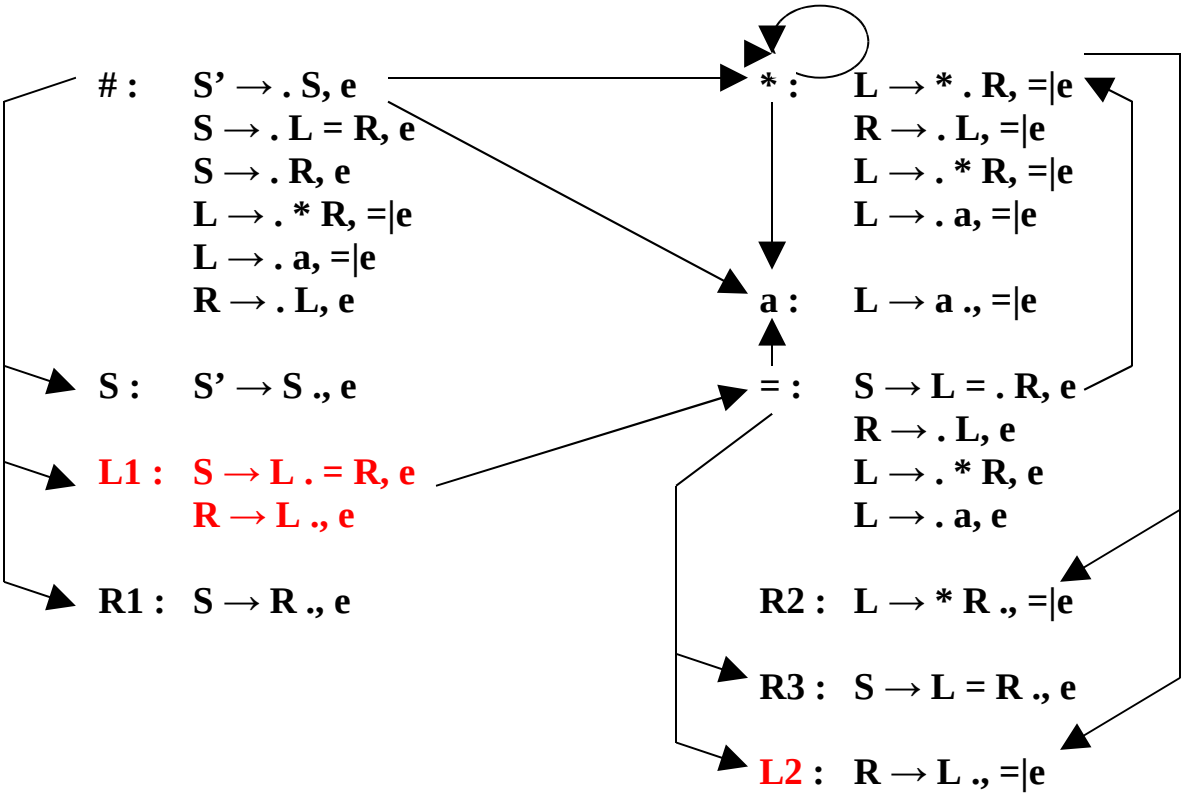


A máme tady problém v L1, která zřetelně indikuje pro „=“ přesun. Pro symboly z FOLLOW(R) by se mělo redukovat podle pravidla R → L Do množiny FOLLOW(R) ale patří i „=“. G_{14} není proto SLR(1). Podíváme se tedy co může následovat za pravou stranou (look ahead) a zkonstruujeme množiny LALR(1) položek.

Rozšířená gramatika G_{14}[S']

0	S' → S
1	S → L = R
2	S → R
3	L → * R
4	L → a
5	R → L

Množiny LALR(1) položek



	a	=	*	e	a	=	*	S	L	R
#	P		P		a		*	S	L1	R1
S				A						
L1		P		5		=				
R1				2						
R2		3		3						
R3				1						
*	P		P		a		*		L2	R2
a		4		4						
=	P		P		a		*		L2	R3
L2		5		5						

Teď zkuste analýzu nějakého řetězce

LR(k) gramatiky

Pro výpočet množin LR(k) položek je třeba jen v algoritmu pro výpočet souboru množin *LALR*(*k*) položek změnit bod 2d) takto:

2d) $\varphi = \varphi \cup \{M_j\}$, GOTO($M_i$, X) = M_j.

To znamená, že vytvořená množina položek M_j se přidá do souboru φ i tehdy, když v φ je již množina položek se stejným jádrem, ale jinými dopředu prohlíženými symboly.

Jestliže algoritmus upravíme tak, že změníme bod 2d), dostaneme algoritmus pro výpočet souboru množin *LR*(*k*) položek.

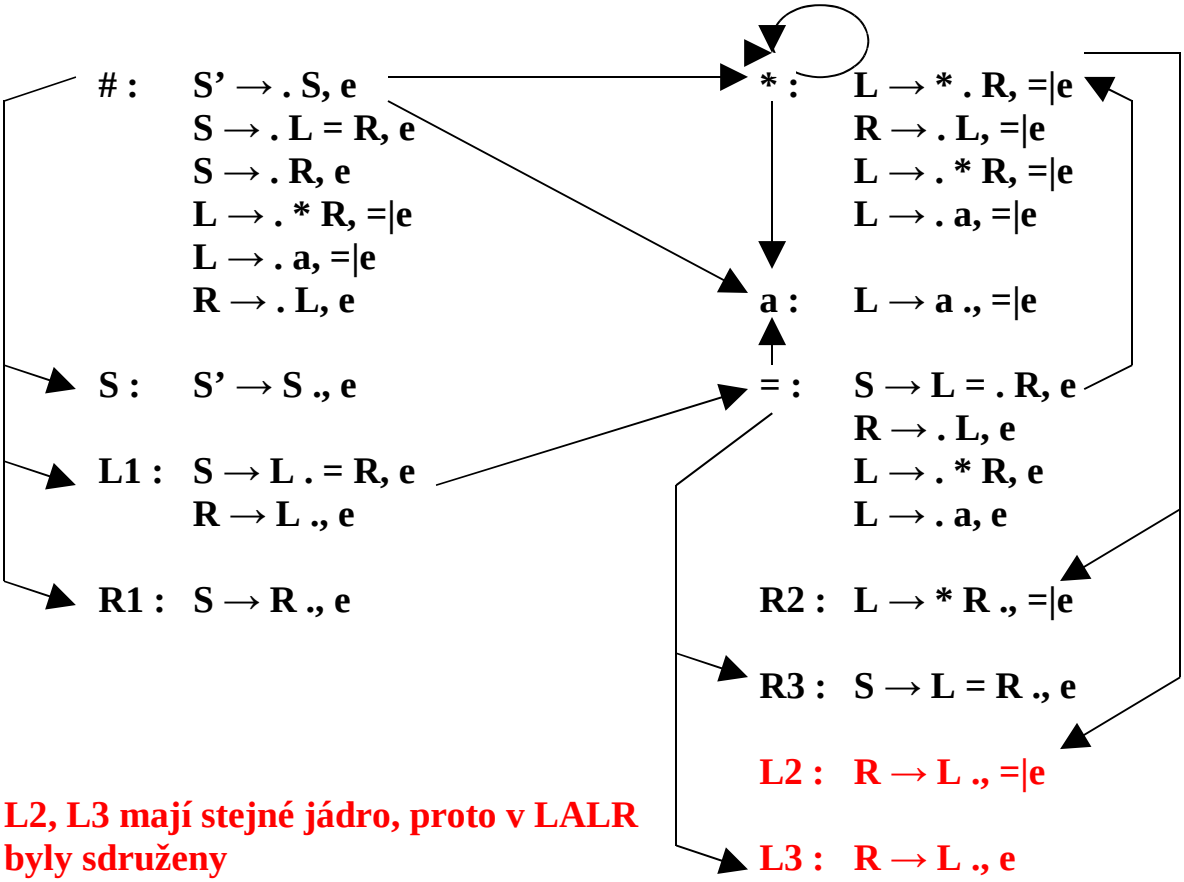
Pro vytvoření tabulky akcí a tabulky přechodů na základě souboru *LR*(*k*) položek můžeme pak použít tytéž algoritmy jako pro *LALR*(*k*) gramatiky.

Definice:

Bezkontextovou gramatiku *G* = (*N, T, P, S*) nazveme *LR*(*k*) gramatikou pro k ≥ 0 právě tehdy, když v souboru množin *LR*(*k*) položek vytvořeného podle upraveného algoritmu pro rozšířenou gramatiku *G'* nejsou žádné konflikty.

Př. G_{14}[S'] řešená jako LR(1)

Množiny LR(1) položek



L2, L3 mají stejné jádro, proto v LALR byly sdruženy

LR(1) rozkladová tabulka

	a	=	*	e	a	=	*	S	L	R
#	P		P		a		*	S	L1	R1
S				A						
L1		P		5		=				
R1				2						
R2		3		3						
R3				1						
*	P		P		a		*		L2	R2
a		4		4						
=	P		P		a		*		L3	R3
L2		5		5						
L3				5						

LR (#, a = * a, -) ├ (# a, = * a, -) ├ (# L_1, = * a, 4) ├
LALR dtto

LR ├ (# L_1 =, * a, 4) ├ (# L_1 = *, a, 4) ├ (# L_1 = * a, e, 4)
LALR dtto

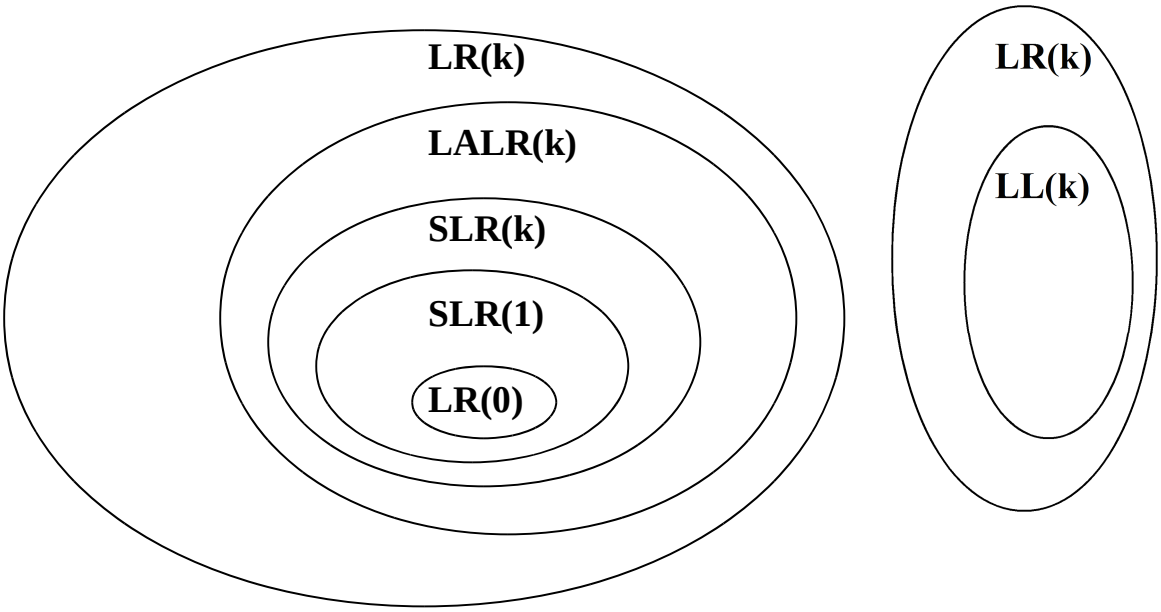
LR ├ (# L_1 = * L_3, e, 4 4) ├ (# L_1 = * R_2, e, 4 4 5) ├
LALR ├ (# L_1 = * L_2, e, 4 4) ├

LR ├ (# L_1 = * R_2, e, 4 4 5) ├ (# L_1 = * L_3, e, 4 4 5 3) ├
LALR dtto

LR ├ (# L_1 = R_3, e, 4 4 5 3 5) ├ (# S, e, 4 4 5 3 5 1)
LALR dtto

Shrnutí

Platí:



Každá LR(k) i každá LL(k) gramatika je jednoznačná

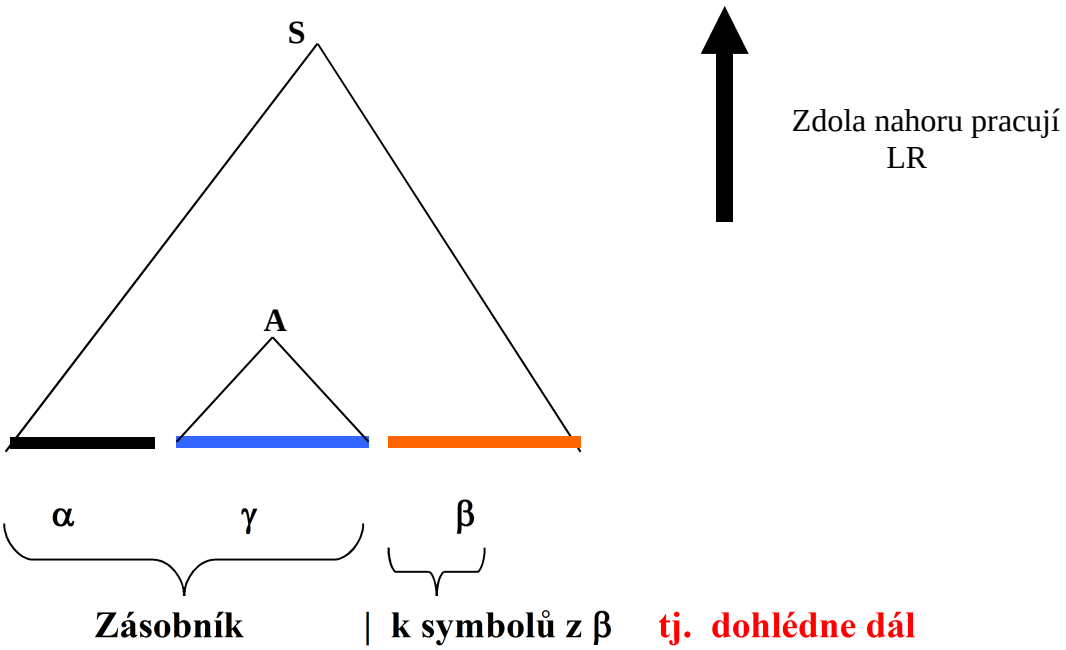
Problém, zda daná gramatika je LR(k) pro zadané k je rozhodnutelný

„ „ „ „ „ „ „ „ libovolné k je nerozhodnutelný

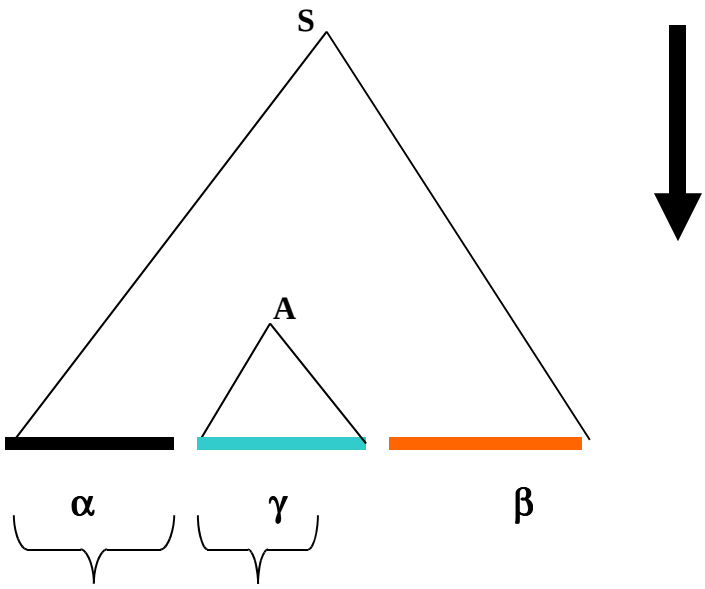
Problém, zda pro jazyk ∃ LR(k) gramatika je nerozhodnutelný

Každou LR(k) gramatiku lze transformovat redukováním FOLLOW na LR(1).

Proč je LR širší třídou než LL?



Je dáno tím, kolik má informace (jak daleko vidí do vstupního textu).



Shora dolů pracují
LL

Zásobník | k symbolů z γ β **proto vidí méně**

Formální překlad

Konečný překladový automat (zopakujme) -KPA **je KA rozšířený o výstup**

KPA = (Q, T, D, δ, q_0, F), kde

Q je množina stavů,
T „ ------ „ vstupních symbolů,
D „ ------ „ výstupních symbolů,
$\delta : Q \times (T \cup \{e\}) \rightarrow 2^{Q \times D^*}$
q_0 je počáteční stav,
$F \subset Q$ je množina koncových stavů.

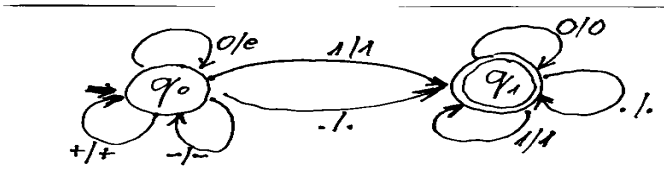
Konfigurace KPA je trojice $(q, x, y) \in Q \times T^* \times D^*$

Stav řadiče
Dosud nepřečtený vstup
Dosud generovaný výstup

Přechody mezi konfiguracemi označíme $\vdash$, $\vdash_p$, $\vdash_*$, $\vdash_+$

Překlad definovaný KPA = { (u, v) : $(q_0, u, e) \vdash_* (r, e, v)$, $r \in F$ }

Př. KPA, který čte binární čísla a vynechává nevýznamové nuly
(Větve ohodnoceny *vstup/výstup*)
Pozn. Přeloží ale i jiné řetězce než bin. čísla (vidíte to?)



<u>Zásobníkový překladový automat</u> ZPA

je ZA rozšířený o výstup resp. KPA s přidaným zásobníkem.

ZPA = (Q, T, D, Γ, δ, q_0, Z_0, F), kde

Q je množina stavů,
T „ ------ „ vstupních symbolů,
D „ ------ „ výstupních symbolů,
Γ „ ------ „ zásobníkových symbolů,
$\delta : Q \times (T \cup \{e\}) \times \Gamma^* \rightarrow 2^{Q \times \Gamma^* \times D^*}$
q_0 je počáteční stav,
Z_0 je dno zásobníku,
$F \subset Q$ je množina koncových stavů.

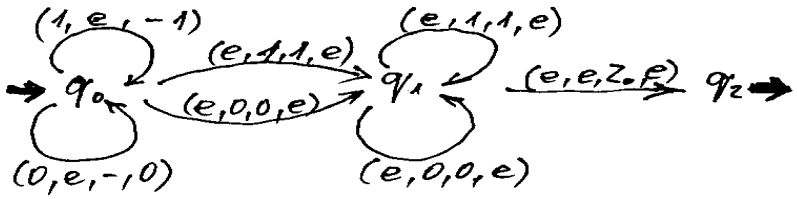
Konfigurace ZPA je $(q, x, y, \alpha) \in Q \times T^* \times D^* \times \Gamma^*$

Obsah zásobníku
Stav řadiče
dosud generovaný výstup
Dosud nepřečtený vstup

Přechod ZA je binární relace $\vdash$, $\vdash_p$, $\vdash_*$, $\vdash_+$ mezi konfiguracemi

Obdobně jako ZA může i ZPA akceptovat při dočtení vstupního řetězce buď přechodem do koncového stavu nebo vyprázdněním zásobníku.

Př. Ohodnocení větví *(vstup, výstup, starý vrchol, nový vrchol)*
Co to vlastně překládá?
(Vkládá vstupní symbol do zásobníku bez ohledu na vrcholový symbol a po přečtení celého vstupního řetězce vypisuje obsah zásobníku až do jeho vyprázdnění.)
Větve jsou ohodnoceny (vstup, výstup, vrchol, nový vrchol)



Takže překládá řetězce z nul a jedniček na jejich zrcadlový obraz

Překladová gramatika

PG = (N, T, D, P, S) kde

D je množina výstupních terminálních symbolů

Evidentně musí platit D ∩ T = Ø

Pro každou PG existuje ekvivalentní ZPA, tzn. L(PG) = L(ZPA)

Konstrukce ekvivalentního ZPA (akceptujícího s prázdným zásob.) k PG
PG = (N, T, D, P, S)
ZPA = ({q}, T, D, N ∪ T ∪ D, δ, q, S, Ø)

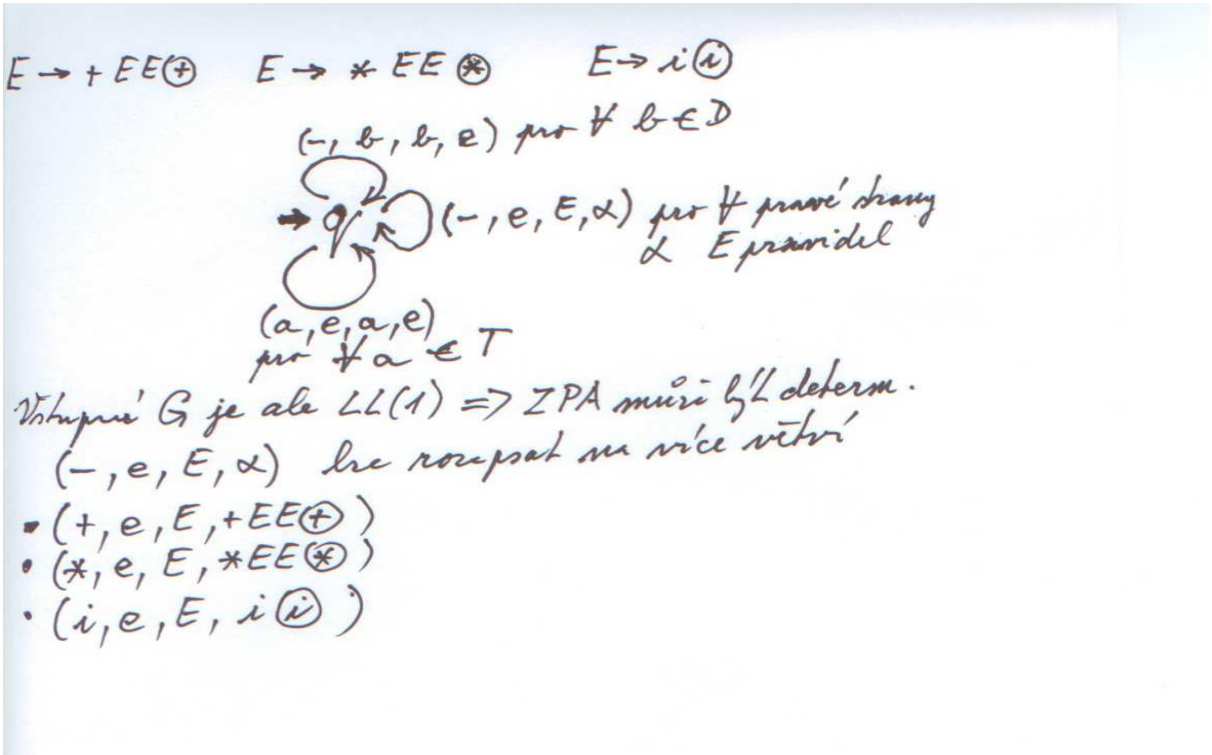
δ : δ (q, -, A) = { (q, α, -) : A → α ∈ P } to je expanze

δ (q, a, a) = { (q, e, e) : pro ∀ a ∈ T } to je srovnání

δ (q, -, b) = { (q, e, b) : pro ∀ b ∈ D } to je výstup
tj. vrchol dá do výstupu

vstup zásobník výstup

Př. Překlad výrazu bez závorek z prefixu do postfixu. Výstupní symboly zde označíme zakroužkováním.



S významem: (vstup, výstup, vrchol, nový vrchol)
Např. je-li na vstupu „+“ a na vrcholu zásobníku E, nedávej nic do výstupu a vrchol E expanduj na „ + E E ⊕ “. Pro „* “ a „ i“je to obdobné.

Atributované překladové gramatiky

APG =
(Překladová gramatika PG, Množina atributů A, Sémantická pravidla SP)

Každému neterminálnímu symbolu X ∈ N je přiřazena množina dědičných atributů I(X) a množina syntetizovaných atributů S(X). S(X) ∩ I(X) = Ø
Každému vstupnímu symbolu je přiřazena množina syntetizovaných atributů a každému výstupnímu symbolu je přiřazena množina dědičných atributů.
Pravidla mají tvar $X_0 \rightarrow X_1 \; X_2 \; \dots \; X_n$, kde $X_0 \in N$
a $X_i \in N \cup T \cup D$ pro $1 \leq i \leq n$

Pro vyhodnocení atributů je nutnou podmínkou aby platilo:
- **Hodnoty děd.atributů počátečního symbolu S jsou zadány,**
- **Hodnoty synt.atributů vstupních terminálních symbolů jsou zadány.**
- **Pro každý dědičný atribut *d* symbolu X_i pravé strany pravidla r tvaru $X_0 \rightarrow X_1 \; X_2 \; \dots \; X_n$ je dáno sémantické pravidlo $d = frdi(a_1, a_2, \dots a_m)$, kde $a_1, a_2, \dots a_m$ jsou atributy symbolů pravidla r,**
- **Pro každý syntetizovaný atribut *s* symbolu X_0 levé strany pravidla r tvaru : $X_0 \rightarrow X_1 \; X_2 \; \dots \; X_n$, je dáno sémantické pravidlo $s = frs0(a_1, a_2, \dots a_m)$, kde $a_1, a_2, \dots a_m$ jsou atributy symbolů pravidla r.**

Jsou-li mezi atributy cyklické závislosti, pak je nelze vyhodnotit

Jednoprůchodový překlad je takový, který dovoluje vyhodnotit všechny atributy v průběhu syntaktické analýzy (jeden průchod synt. stromem.

Pro jednoprůchodový překlad je nutné aby platilo, že hodnoty atributů každého symbolu závisí pouze na atributech již zpracovaných symbolů (s vyhodnocenými atributy). Takovou gramatiku zveme L-atributovanou.
Platí pro každé její pravidlo (r) : $X_0 \rightarrow X_1 \; X_2 \; \dots \; X_n$
1) **Pro každý dědičný atribut *d* symbolu X_i pravé strany $d=frdi(a_1,a_2,\dots a_m)$, kde $a_1, a_2, \dots a_m$ jsou buď dědičné atributy symbolu X_0 nebo dědičné a syntetizované atributy symbolů $X_1,X_2,\dots X_{i-1}$**
2) **Pro každý syntetizovaný atribut *s* levostranného symbolu X_0 je $s=frs0(a_1, a_2, \dots a_m)$, kde $a_1, a_2, \dots a_m$ jsou buď dědičné atributy symbolu X_0 nebo dědičné a syntetizované atributy symbolů z pravé strany pravidla.**

Překlad při LL analýze

Pro LL analýzu musí být splněno:

1. **vstupní gramatika (gram. bez výstupních symbolů) splňuje LL podmínky.**
2. **PG je sémanticky jednoznačná (pravidla se nesmí odlišovat pouze výstupními symboly).**
3. **je L-atributovaná**

Konstrukce rozkladové tabulky je stejná (s uvážením vstupního homomorfismu) jako akceptačního automatu.

Postup zpracování vstupního řetězce je obdobný jako u akceptačního automatu s těmito rozdíly:

1. **Do zásobníku jsou ukládány symboly včetně jejich atributů**
2. **Je-li na vrcholu zásobníku výstupní symbol, je přenesen i s atributy do výstupu**
3. **Při expanzi se provedou sémantické akce příslušného pravidla**

Př. Překlad přiřazovacího příkazu do postfixových instrukcí

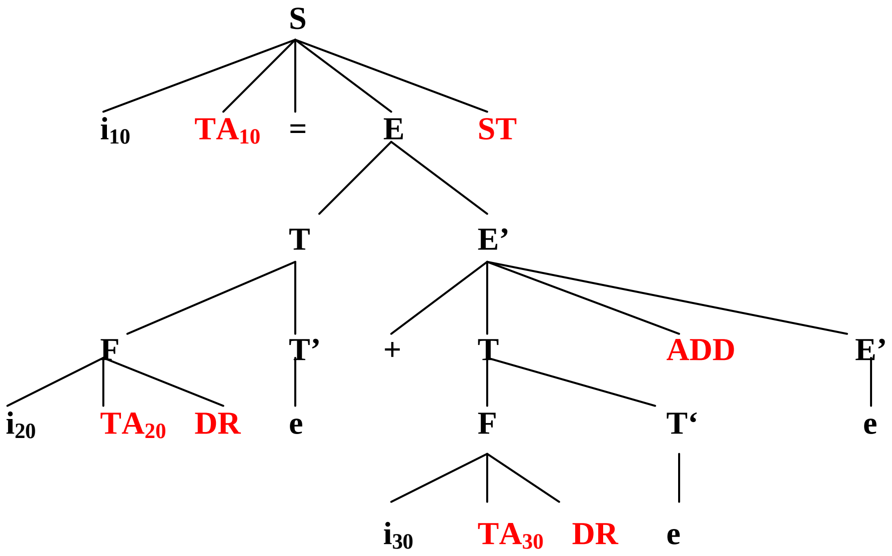
APG= (překl.gr.pro přiřazovací příkaz, s výst. symboly *TA, DR, ST, ADD, MUL*
Syntetizovaný atribut adr,
Sémantická pravidla)

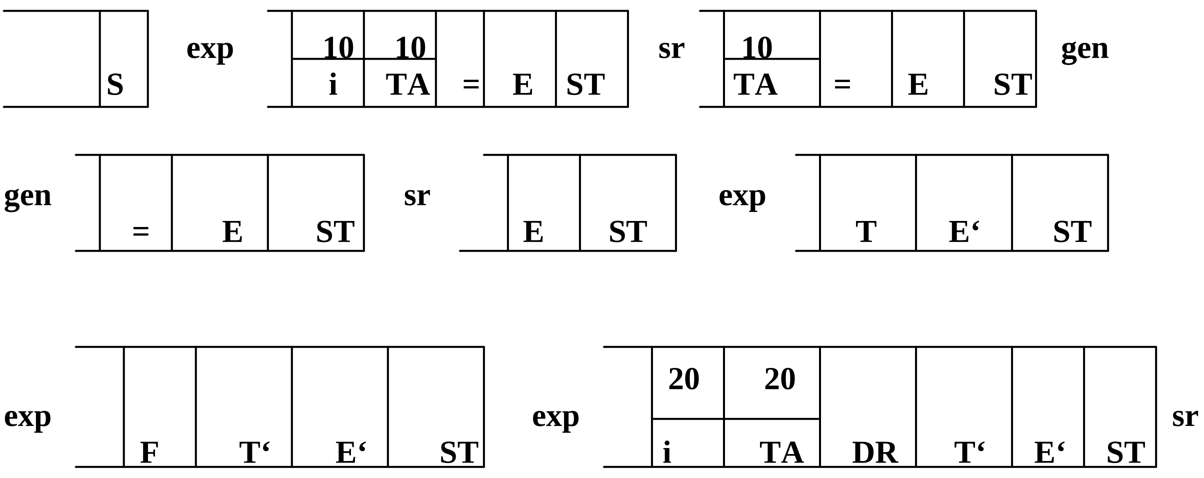
Gramatická pravidla	sémantická pravidla
1. S → i *TA* = E *ST*	***TA*.adr ← i.adr**
2. E → T E'	
3. E' → + T *ADD* E'	
4. E' → e	
5. T → F T'	
6. T' → * F *MUL* T'	
7. T' → e	
8. F → (E)	
9. F → i *TA* *DR*	***TA*.adr ← i.adr**

Překlad probíhá současně s analýzou:

	=	i	+	*	(	)	e
S		**1**					
E		**2**			**2**		
E'			**3**			**4**	**4**
T		**5**			**5**		
T'			**7**	**6**		**7**	**7**
F		**9**			**8**		

Např. i_{10} = i_{20} + i_{30}





sr …

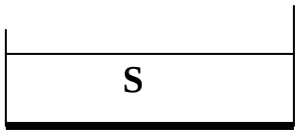
př.) Překlad přiřazovacího příkazu do čtveřic

Zavedeme pomocnou fci NPP pro generování nové pomocné proměnné, výstupní symboly ASS, ADD, MUL a atributy

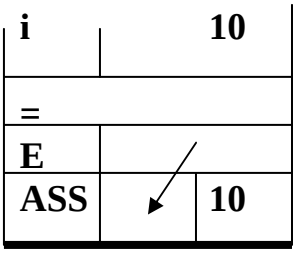
Symb.	Atributy	Význam
E	E.ukaz	ukazatel na místo kam dosadit adresu s hodn. E
ASS	ASS.p ASS.l	adresa pravé a levé strany přiřazení
E’	E’.adr E’.ukaz	adresa s hodnotou E’, ukazatel kde má být E’ použita
ADD	ADD.l ADD.p ADD.v	adresy levého, pravého a výsledkového operandu
MUL	“----------“ ---------“	“----------“ ---------“
T	T.ukaz	ukazatel kam dosadit adresu s hodnotou T
T’	T’.adr T’.ukaz	adresa s hodnotou T’, ukazatel kde má být T’ použita
F	F.ukaz	ukazatel kam dosadit adresu s hodnotou F
i	i.adr	adresa přidělená identifikátoru i

syntaktická pravidla	sémantická pravidla
1) S → i = E ASS	ASS.l ← i.adr ; E.ukaz ← ukazatel na ASS.p
2) E → T E‘	E‘.ukaz ← E.ukaz ; T.ukaz ← ukazatel na E‘.adr
3) E‘ → + T ADD E’^{1}	PP ← NPP ; ADD.v ← PP ; ADD.l ← E’.adr ; E’^{1}.adr←PP ; T.ukaz←ukazatel na ADD.p ; E’^{1}.ukaz←E’.ukaz
4) E‘ → e	kam ukazuje E’.ukaz ← E’.adr
5) T → F T’	T’.ukaz ← T.ukaz ; F.ukaz ← ukazatel na T’.adr
6) T’ → * F MUL T’^{1}	PP ← NPP ; MUL.v ← PP ; MUL.l ← T’.adr ; T’^{1}.adr←PP ; F.ukaz←ukazatel na MUL.p ; T’^{1}.ukaz←T’.ukaz
7) T’ → e	kam ukazuje T’.ukaz ← T’.adr
8) F → (E)	E.ukaz = F.ukaz
9) F → i	kam ukazuje F.ukaz ← i.adr

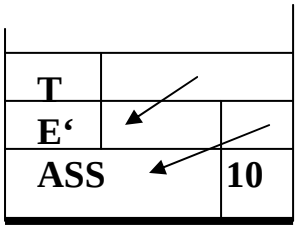
Př. $i_{10} = i_{20} + i_{30}$



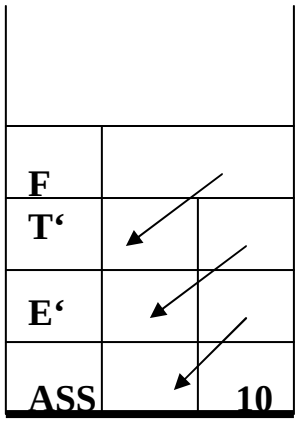
ex 1



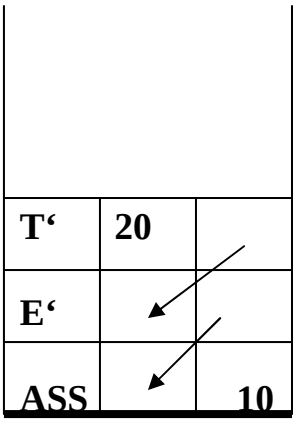
sr,sr,ex2



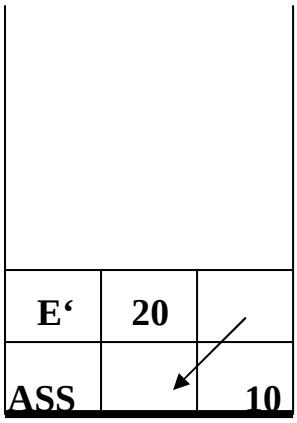
ex 5



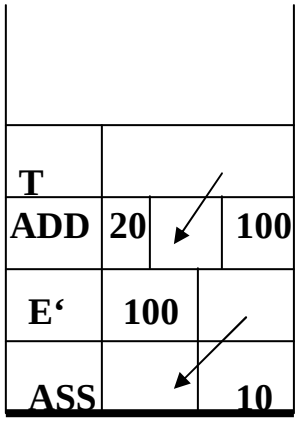
ex9,sr



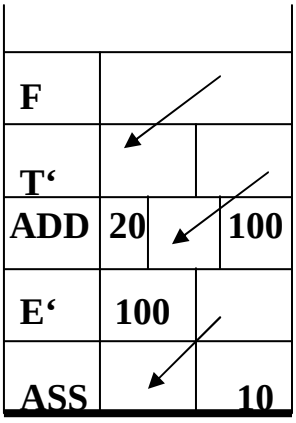
ex7



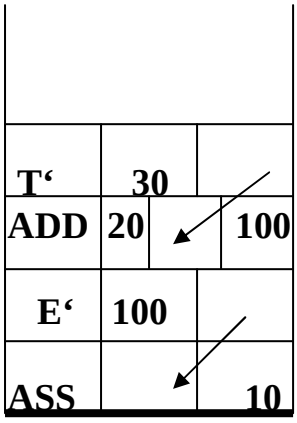
ex 3, sr



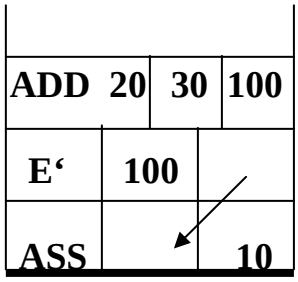
ex5



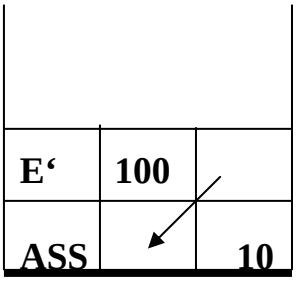
ex9,sr



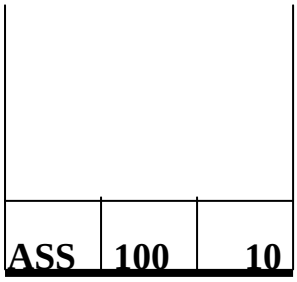
ex 7



gen



ex4



gen

Překlad při LR analýze

Př. Překlad přiřazovacího příkazu zdola nahoru do postfixové notace

	=	i	+	*	(	)	e	E	T	F	LS	+	*	(	)	=	i	S
#		P									LS						i1	S
E1			P				7					+						
T1			2	P		2	2						*					
F1			4	4		4	4											
i2			6	6		6	6											
(		P			P			E2	T1	F1				(			i2	
+		P			P				T2	F1				(			i2	
*		P			P					F2				(			i2	
E2			P			P						+			)			
T2			1	P		1	1						*					
F2			3	3		3	3											
)			5	5		5	5											
LS	P															=		
=		P			P			E1	T1	F1				(			i2	
S							A											
i1	8																	

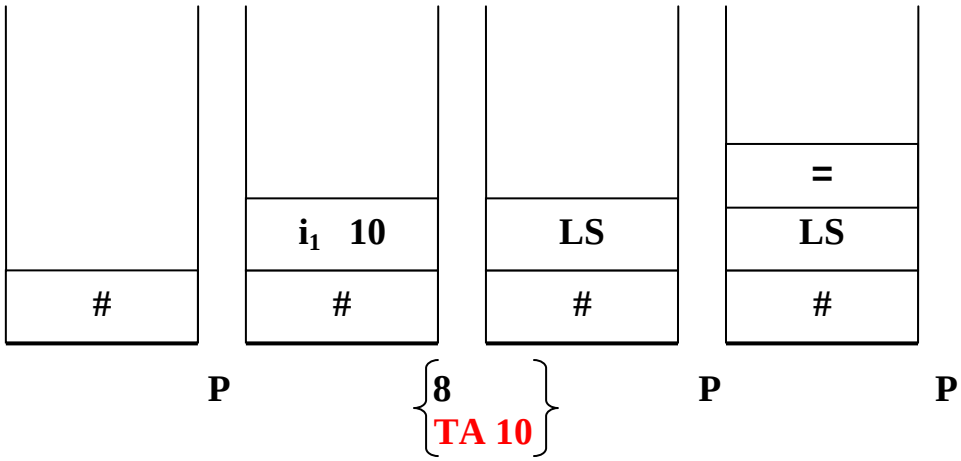
Gramatika rozšířená o sémantické akce prováděné při redukcích:

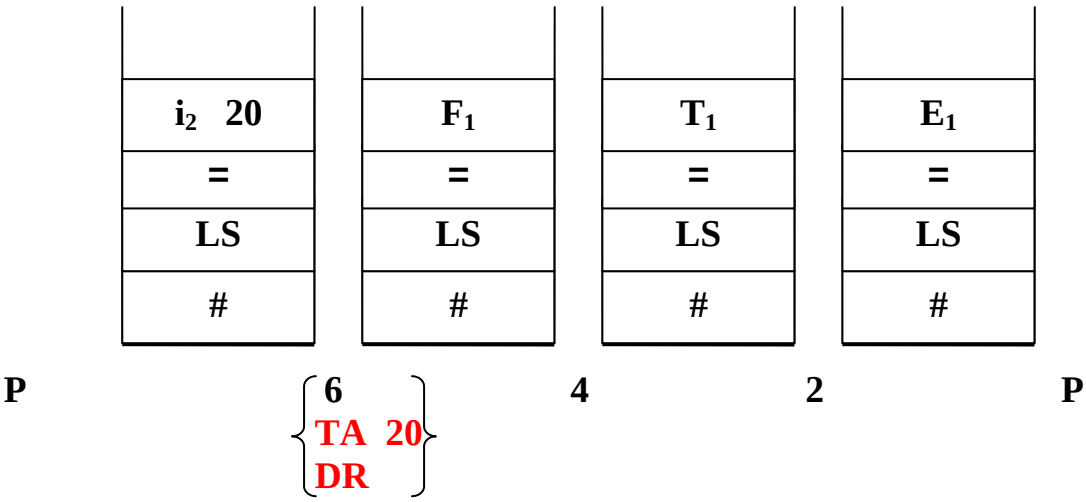
0) **S'** → **S**
1) **E** → **E + T** **GEN(ADD)**
2) **E** → **T**
3) **T** → **T * F** **GEN(MUL)**
4) **T** → **F**
5) **F** → **(E)**
6) **F** → **i** **GEN(TA i.adr) ; GEN(DR)**
7) **S** → **LS = E** **GEN(ST)**
8) **LS** → **i** GEN(TA i.adr)

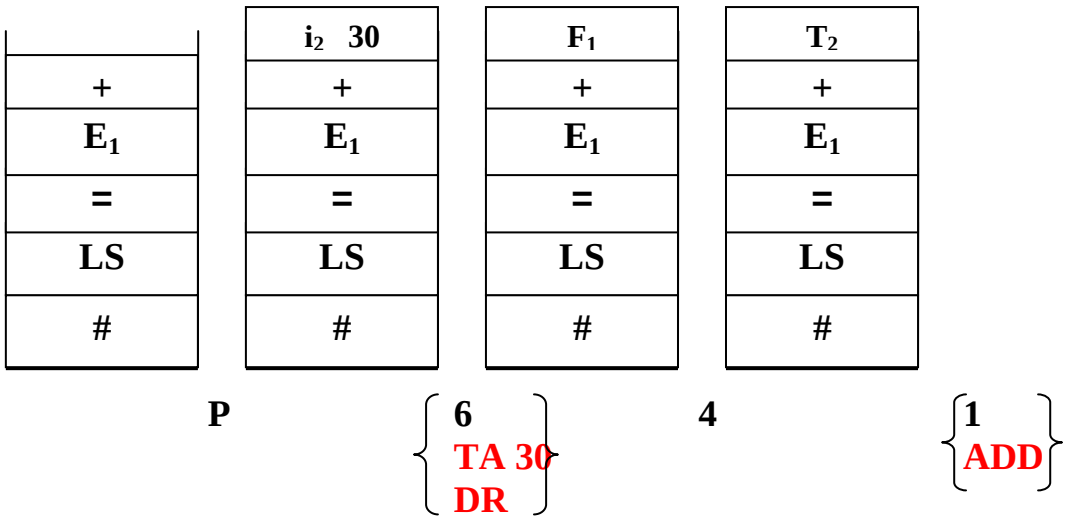
Tak jednoduché je to v případech, když APG používá jen syntetizované atributy a výstupní symboly jsou jen na koncích pravých stran pravidel. Taková APG se nazývá S-atributovaná.
Ve formalismu APG místo akcí generování zavedeme výstupní symboly

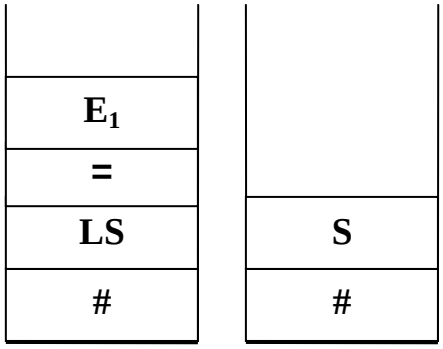
	syntax			**sémantika**
0)	**S'**	→	**S**	
1)	**E**	→	**E + T ADD**	
2)	**E**	→	**T**	
3)	**T**	→	**T * F MUL**	
4)	**T**	→	**F**	
5)	**F**	→	**(E)**	
6)	**F**	→	**i TA DR**	**TA .adr = i.adr**
7)	**S**	→	**LS = E ST**	
8)	**LS**	→	**i TA**	**TA .adr = i.adr**

Př. $i_{10} = i_{20} + i_{30}$





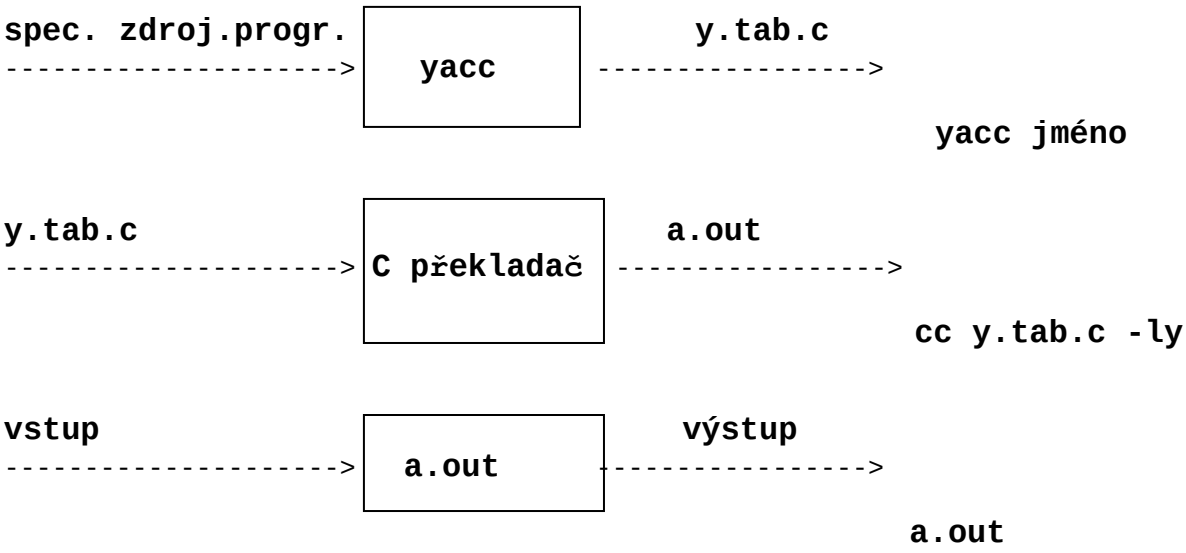




7
ST

Akcept

YACC



Tvar zdrojového programu:

```
deklarace
%%
překladová pravidla
%%
pomocné C - programy
```

Tvar překladových pravidel:

```
<levá strana> : <pravá strana 1> {sémantická akce 1}
              | <pravá strana 2> {sémantická akce 2}
                        ...
              | <pravá strana n> {sémantická akce n}
```

Deklarace obsahují:

1. Běžné C deklarace oddělené %{ a %}
2. Deklarace gramatických symbolů, pojmenovávajících terminální symboly

Metasymboly:

```
      '  ' omezovače terminálního symbolu
     $$   hodnota atributu levostranného symbolu
     $i   hodnota atributu i-tého pravostranného symbolu
```

Pomocné procedury:

Musí vždy uvést lexikální analyzátor yylex().
Lze jej nahradit # include "lex.yy.c"
Další pomocné procedury jsou fakultativní

Příklad kalkulačky.
Čte aritmetický výraz a vyhodnocuje jej

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
line      :      expr '\n'            {printf("%d\n", $1);}
          ;
expr      :      expr '+' term        {$$ = $1 + $3; }
          |      term
          ;
term      :      term '*' factor      {$$ = $1 * $3; }
          |      factor
          ;
factor    :      '(' expr ')'         {$$ = $2; }
          |      DIGIT
          ;
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
       yylval = c-'0';
       return  DIGIT;
    }
    return c;
}
```

Voláním yacc -v jmenosouboru
vytvoří navíc soubor y.out

```
state 0
     $accept : _line $end

     DIGIT  shift 6
     (  shift 5
     .  error

     line  goto 1
     expr  goto 2
     term  goto 3
     factor  goto 4


state 1
     $accept :  line_$end

     $end  accept
     .  error


state 2
     line :  expr_\n
     expr :  expr_+ term

     \n  shift 7
     +  shift 8
     .  error


state 3
     expr :  term_    (3)
     term :  term_* factor

     *  shift 9
     .  reduce 3


state 4
     term :  factor_    (5)

     .  reduce 5
```

```
state 5
     factor :  (_expr )

     DIGIT  shift 6
     (  shift 5
     .  error

     expr  goto 10
     term  goto 3
     factor  goto 4

state 6
     factor :  DIGIT_    (7)

     .  reduce 7


state 7
     line :  expr \n_    (1)

     .  reduce 1


state 8
     expr :  expr +_term

     DIGIT  shift 6
     (  shift 5
     .  error

     term  goto 11
     factor  goto 4

state 9
     term :  term *_factor

     DIGIT  shift 6
     (  shift 5
     .  error

     factor  goto 12

state 10
     expr :  expr_+ term
     factor :  ( expr_)

     +  shift 8
     )  shift 13
     .  error
```

```
state 11
     expr :  expr + term_    (2)
     term :  term_* factor

     *  shift 9
     .  reduce 2



state 12
     term :  term * factor_    (4)

     .  reduce 4


state 13
     factor :  ( expr )_    (6)

     .  reduce 6


8/200 terminals, 4/300 nonterminals
8/600 grammar rules, 14/750 states
0 shift/reduce, 0 reduce/reduce conflicts reported
8/350 working sets used
memory: states,etc. 72/12000, parser 9/12000
9/600 distinct lookahead sets
4 extra closures
14 shift entries, 1 exceptions
7 goto entries
3 entries saved by goto default
Optimizer space used: input 40/12000, output 218/12000
218 table entries, 204 zero
maximum spread: 257, maximum offset: 42
```

Příklad kalkulačky pro reálná čísla
(používá nejednoznačnou gramatiku)

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /*double typ pro Yacc stack*/
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
lines : lines expr '\n'  {printf("%g\n", $2); }
      | lines '\n'
      | /* e */
      ;
expr  : expr '+' expr            { $$ = $1 + $3; }
      | expr '-' expr            { $$ = $1 - $3; }
      | expr '*' expr            { $$ = $1 * $3; }
      | expr '/' expr            { $$ = $1 / $3; }
      | '(' expr ')'             { $$ = $2;      }
      | '-' expr %prec UMINUS    { $$ = - $2;    }
      | NUMBER
      ;
%%
yylex() {
      int c;
      while ( ( c = getchar() ) == ' ');
      if ( ( c == '.') || (isdigit(c)) ) {
            ungetc(c, stdin);
            scanf("%lf", &yylval);
            return NUMBER;
      }
      return c;
}
```

LALR algoritmus s konflikty řeší YACC takto:

Konflikt redukce-redukce řeší výběrem pravidla dle pořadí jejich uvedení.
Konflikt redukce-přesun řeší upřednostněním přesunu.