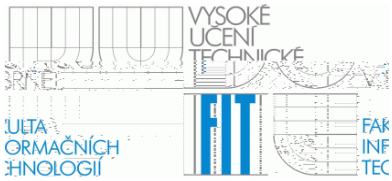


Formální jazyky a překladače

IFJ

Studijní opora

Alexander Meduna
Roman Lukáš

Verze: 1.2006+revize 2009-2015

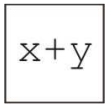
Tento učební text vznikl za podpory projektu „Zvýšení konkurenceschopnosti IT odborníků – absolventů pro Evropský trh práce“, reg. č. CZ.04.1.03/3.2.15.1/0003. Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky.

Obsah

0. Poznámka k těmto oporám

Tento text slouží jako studijní opora předmětu Formální jazyky a překladače (IFJ), který je vyučován ve druhém ročníku bakalářského studijního programu na FIT VUT v Brně. Je v něm pokryta převážná část přednášek tohoto předmětu. Pro lepší orientaci studentů v textu jsou v něm použity následující piktogramy:

…... definice



…... příklad

…..... cvičení

Prosíme omluvte sníženou typografickou kvalitu textu. Text postupně prochází typografickou úpravou.

1. Úvod do teorie překladačů

Prakticky každý programátor dnes programuje v některém vyšším programovacím jazyce (např. PASCAL či C). Procesor počítače ovšem provádí programy ve svém strojovém jazyce. Má-li být tedy programátorův program počítačem zpracován, musí být nejprve transformován na funkčně ekvivalentní program ve strojovém jazyce, což provádí systémový program, který se nazývá kompilátor.
Obecně tedy můžeme říci, že kompilátor je program, který na vstupu přijímá zdrojový program zapsaný ve zdrojovém jazyce, aby k němu na výstupu generoval funkčně ekvivalentní cílový program v cílovém jazyce. Transformace, kterou kompilátor provádí, se nazývá překlad (ze zdrojového do cílového jazyka).

Překlad zdrojového programu zpravidla probíhá v šesti hlavních fázích:

1. lexikální analýza,
2. syntaktická analýza,
3. sémantická analýza
4. generování vnitřní formy programu
5. optimalizace
6. generování cílového programu.

Z uvedených názvů hlavních fází překladu jsou odvozeny i názvy jednotlivých částí kompilátoru, které příslušné fáze provádějí: lexikální analyzátor, syntaktický analyzátor a generátor kódu.

Lexikální analyzátor čte zdrojový program a překládá jej na řetěz lexikálních symbolů, což jsou jednotlivé elementy zdrojového jazyka, např. identifikátory, klíčová slova apod. Výstup lexikálního analyzátoru je vstupem pro **syntaktický analyzátor**, jehož úkolem je zjistit, zda je zdrojový program syntakticky správně zapsán, tzn. zda řetěz lexikálních symbolů patří do zdrojového jazyka. Pokud tomu tak skutečně je, pak výstupem syntaktického analyzátoru je tzv. derivační strom, který reprezentuje syntaktickou strukturu zdrojového programu. Sémantický analyzátor provádí kontrolu nejrůznějších sémantických aspektů programu, jako např. deklarovanost proměnných.

Na základě syntaktické struktury zdrojového programu vytváří **generátor kódu** cílový program. Generování kódu se přitom zpravidla samo rozpadá, jak jsme již uvedli, do třech samostatných fází: **generování vnitřní formy programu, optimalizace, generování cílového programu**. Syntaktický strom, který je zpravidla výstupem ze syntaktického analyzátoru, se nejprve přeloží na program zapsaný v tzv. vnitřní formě, která zpravidla umožňuje provést poměrně jednoduchým způsobem optimalizaci zdrojového programu. Takovýto optimalizovaný program ve vnitřní formě je nakonec přeložen generátorem vnitřního programu na funkčně ekvivalentní program v cílovém jazyce.

Jak jsme tedy viděli, proces překladu daného zdrojového programu se sám uskutečňuje jako série na sebe navazujících dílčích překladů,

přičemž výstup z jednoho z těchto překladů se vzápětí stává vstupem následujícího.

Již z tohoto stručného úvodu by mělo být jasné, že ústředními pojmy této práce bude (formální) jazyk a jeho překlad. Formálními jazyky a prostředky pro jejich definici se budeme zabývat v následující kapitole 2. Kapitola 3, která tvoří jádro celé práce, je pak již přímo věnována překladu a základním možnostem jeho specifikace. Vyložíme si v ní všechny fundamentální metody konstrukce jednotlivých částí kompilátoru, jak jsme si je v krátkosti uvedli výše.

2. Gramatiky a automaty

2.1 Abecedy, slova, jazyky

DEF

Definice 2.1.1

Abeceda je libovolná neprázdná konečná množina. Prvky abecedy nazýváme symboly.
Tak např. {0, 1} je (binární) abeceda.

DEF

Definice 2.1.2

Buď abeceda. Slovo nad je konečná posloupnost symbolů z . Prázdnou posloupnost budeme nazývat prázdné slovo a označovat jej ε.

V zápisu slov budeme pro jednoduchost vynechávat čárky. Tedy ku příkladu slovo 0, 0, 1, 1 nad {0, 1} budeme jednoduše psát 0011.

V literatuře se někdy místo termínu slovo užívá názvu řetěz popř. řetězec. Všechny tyto pojmy budeme proto používat jako synonyma.

Všimněme si, že prázdné slovo je slovo nad libovolnou abecedou.

Buď = 1 … n slovo nad abecedou , i ∈ , 1 ≤ i ≤ n, pro nějaké n ≥ 1. Pak délku tohoto slova budeme označovat | | = n. Délka prázdného slova je 0, tzn. |ε| = 0.

Mějme slova ε, 0, 111 nad abecedou {0, 1}. Jejich délka je: |ε| = 0, |0| = 1, |111| = 3.

Buď u, dvě slova nad abecedou . Pak slovo u nazýváme zřetězení slov u a .

Zřetězením slov "pra" a "děd" nad českou abecedou je slovo praděd. Jestliže zřetězíme slova 0 a 110 nad binární abecedou získáváme slovo 0110.

Poznamenejme, že pro každé slovo x platí εx = xε = x. Speciálně pro x = ε: εε = ε.

Buď abeceda. Pak pro libovolná slova x, , nad :

$$(x\ )\ = x(\ \ ) = x\ \ ,$$

kde závorky specifikují, které zřetězení se provede dříve. Jinými slovy, zřetězení je asociativní.

Dále, pro každá dvě slova x, y zřejmě platí:

$$|x\ | = |x| + |\ |$$

a tedy speciálně

$$|x\varepsilon| = |x| + |\varepsilon| = |x|.$$

Pomocí zřetězení definujeme mocninu daného slova x.
Buď x slovo nad nějakou abecedou V. Pak i-tou mocninu slova x budeme označovat x^i a definovat ji:

(1) $x^i = \varepsilon$ pro $i = 0$;
(2) $x^i = xx^{i-1}$ pro každé $i \geq 1$.

Zřejmě pro každé slovo x:

$$|x^i| = i \cdot |x|,$$
$$x^i x^j = x^{i+j} = x^j x^i$$

pro každé $i, j \geq 0$.

x+y

Příklad 2.1.3
Buď ε, , , slova nad abecedou { , }. Pak jejich délka je

|ε| = 0,
| | = 1,
| | = 2,
| | = 3.

Zřetězením slov a získáme slovo .
Jehož délka je | | + | | = 1 + 3 = 4. Všimněme si, že slovo je také druhá mocnina slova , tj. ( )^2 = , pro které platí | | = |( )^2| = 2. | | = 2 · 2 = 4.

Buď V abeceda, x, slova nad V. Říkáme, že x je <u>prefix</u> (resp. <u>sufix</u>) <u>slova</u> , existuje-li slovo nad V tak, že platí x = (resp. x =).
Jestliže nadto ≠ $x \neq \varepsilon$, pak x je <u>vlastní podslovo slova</u> .

Každý prefix a sufix daného slova je zřejmě jeho podslovem. Toto tvrzení však neplatí naopak: *off* je podslovo anglického slova *coffee*, ale není jeho sufix či prefix.

x+y

Příklad 2.1.4
Uvažujme slovo *on* nad českou abecedou. Všechny jeho prefixy jsou:

ε, *o*, *on*, *on* .

Slova , *n* jsou všechny vlastní sufixy slova *on* – slovo ε, *on* jsou sice také sufixy tohoto slova, ale nikoliv vlastní. A slova *o*, *n*, , *on*, *n* , *on* , ε jsou všechna podslova slova *on* , přičemž poslední dvě nejsou vlastní.

Pro každé slovo x nad nějakou abecedou V definujeme slovo x^R:

(1) $x^R = x$, jestliže $x = \varepsilon$;
(2) $x^R = {}^R$, jestliže $x =$ pro nějaký řetězec $\in V^*$, a $\in V$.

Slovo x^R nazýváme <u>reverzí slova</u> x.

Reverzi slova x získáme tedy jednoduše tak, že jeho symboly napíšeme v obráceném pořadí. Ku příkladu reverzi českého slova x = *o k r o c n í* je slovo x^R = *í n o c r k o*.

Snadno se přesvědčíme, že pro libovolná dvě slova x, y platí: $(xy)^R = y^R x^R$.

Přejděme nyní k definici jazyka nad danou abecedou Σ. Nechť Σ^* označuje množinu všech řetězců nad Σ a $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.

DEF

Definice 2.1.5

Buď Σ abeceda a nechť $L \subseteq \Sigma^*$. Pak L je jazyk nad abecedou Σ.

Prázdná množina $\emptyset$ a $\{\varepsilon\}$ jsou tedy jazyky nad každou abecedou. Přitom ovšem platí

$$\emptyset \neq \{\varepsilon\}$$

neboť jazyk $\{\varepsilon\}$ obsahuje jedno (prázdné) slovo, zatímco jazyk $\emptyset$ neobsahuje žádné slovo. Obsahuje-li jazyk konečný počet slov, nazýváme jej konečný. Tak např. { , , } je konečný jazyk nad { }. Každý jazyk, který není konečný, nazýváme nekonečný. Tak např. { }* je nekonečný jazyk nad { }.

Poněvadž jazyky jsou množiny (slov), lze s nimi provádět množinové operace jako například sjednocení, průnik, rozdíl a doplněk.

Mějme dva jazyky $L_1 \subseteq \Sigma^*$ a $L_2 \subseteq \Sigma^*$. Pak

$$L_1 L_2 = \{x : x \in L_1, \; \in L_2\}$$

nazýváme zřetězením jazyků L_1 a L_2.

Pro libovolné jazyky L_1, L_2, L_3 zřejmě platí:

$$L_1 (L_2 L_3) = (L_1 L_2) L_3 = L_1 L_2 L_3$$
$$\{\varepsilon\} L_1 = L_1 \{\varepsilon\} = L_1$$
$$\emptyset L_1 = L_1 \emptyset = \emptyset$$
$$L_1 (L_2 \cup L_3) = L_1 L_2 \cup L_1 L_3$$
$$(L_1 \cup L_2) L_3 = (L_1 L_3 \cup L_2 L_3)$$

Pro daný jazyk $L \subseteq \Sigma^*$ a celé číslo $i \geq 0$ definujeme *i*-tou mocninu L, označíme ji L^i, takto:

1) $L^i = \{\varepsilon\}$, jestliže $i = 0$;
2) $L^i = L^{i-1}$, jestliže $i \geq 1$.

Iterací jazyka L budeme označovat L^* a definovat:

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Kladnou iteraci budeme označovat $^+$, definujeme:

$$^+ = \bigcup_{i=1}^{\infty} {}^i$$

Přímo z definice vyplývá:

$$\varnothing^+ = \varnothing$$
$$\varnothing^* = \{\varepsilon\}^* = \{\varepsilon\}^+ = \{\varepsilon\}$$

Pro daný jazyk :

$$^+ = {}^+$$
$$^* = {}^* = (\ ^+)^+ = {}^+$$
$$(\ ^*)^+ = (\ ^*)^* = (\ ^+)^* = {}^*$$

a pro libovolné dva jazyky $_1$ a $_2$:

$$_1 \subseteq {}_2 \text{ implikuje } {}_1^+ \subseteq {}_2^+$$
$$_1 \subseteq {}_2 \text{ implikuje } {}_1^* \subseteq {}_2^*$$

Reverzi jazyka , označme si ji $^\leftarrow$, definujeme:

$$^\leftarrow = \{x^\leftarrow : x \in \ \}$$

Snadno se přesvědčíme, že:

$$(\ _1)^\leftarrow = {}_1$$
$$(\ _1 \cup \ _2)^\leftarrow = {}_1^\leftarrow \cup {}_2^\leftarrow$$
$$(\ _1 \cap \ _2)^\leftarrow = {}_1^\leftarrow \cap {}_2^\leftarrow$$
$$(\ _1 \ _2)^\leftarrow = {}_2^\leftarrow \ {}_1^\leftarrow$$
$$(\ _1^+)^\leftarrow = (\ _1^\leftarrow)^+$$
$$(\ _1^*)^\leftarrow = (L_1^\leftarrow)^*$$

pro každé dva jazyky $_1$, $_2$.

x+y

Příklad 2.1.6

Mějme dva jazyky *A*, *B* nad českou abecedou, oba obsahují jedno slovo:

$$A = \{pr\ \}, B = \{\ \check{e}\ \}.$$

Pak např.

$$AB = \{pr\ \ \check{e}\ \},$$
$$B^0 = \{\varepsilon\},$$
$$A^+ = \bigcup_{i=1}^{\infty} (pr\ )^i,$$
$$B^\leftarrow = \{\ \check{e}\ \}$$

a

$$A^*B = \{\ \check{e}\ , pr\ \ \check{e}\ , pr\ pr\ \ \check{e}\ , pr\ pr\ pr\ \ \check{e}\ , \ldots\}$$

2.2 Konečná reprezentace jazyků

Chceme-li v praxi s formálními jazyky pracovat, potřebujeme konečné prostředky pro jejich specifikaci. Konečné jazyky lze jednoduše popsat výčtem jejich slov. Takováto specifikace je však nemožná pro nekonečné jazyky (mezi které mj. patří prakticky všechny programovací jazyky). Proto se zde budeme zabývat jinými prostředky pro specifikaci jazyka: gramatikami a automaty. Jedná se totiž o dva základní typy konečné reprezentace (nejen konečných, ale i nekonečných) jazyků.

Podstatou každé gramatiky je konečná množina pravidel, pomocí kterých lze generovat daný jazyk. Jednotlivá slova tohoto jazyka se vytváří postupným přepisováním řetězců podle těchto pravidel.

I automat definuje jazyk pomocí konečných prostředků. Je to vlastně algoritmus, který pro libovolné slovo nad vstupní abecedou rozhodne, zda patří, či nepatří do daného jazyka.

V následujících odstavcích probereme detailně ty typy gramatik a automatů, které mají stěžejní význam (jak později uvidíme) při konstrukci kompilátorů.

2.3 Chomského klasifikace gramatik

Začneme obecnou definicí gramatiky.

DEF

Definice 2.3.1.
Generativní gramatika je čtveřice $G = (N, T, P, S)$, kde

1) N je abeceda
2) T je abeceda,
   $N \cap T = \emptyset$;
3) $P \subseteq (N \cup T)^* (N \cup T)^* \times (N \cup T)^*$
   je konečná množina;
4) $S \in N$

N – nazýváme abeceda neterminálů, T – abeceda terminálů, P – konečná množina pravidel a S je tzv. počáteční symbol.

Pravidlo $(x, y) \in P$ obvykle zapisujeme: $x \to y$.

Mějme slova $u \in (N \cup T)^+$, $v \in (N \cup T)^*$. Řekneme, že u přímo derivuje v (u v), což zapisujeme $u \Rightarrow v$, jestliže existují slova u_1, u_2, x, $y \in (N \cup T)^*$ tak, že $u = u_1 x u_2$, $v = u_1 y u_2$, $x \to y \in P$. Řekneme, že slovo u derivuje v, symbolicky $u \Rightarrow^* v$, právě když existuje posloupnost $u_1, ..., u_n$, $n \geq 1$ slov nad $(N \cup T)$ tak, že $u = u_1 \Rightarrow u_2 \Rightarrow ... \Rightarrow u_n = v$. Jazyk generovaný gramatikou G, označme si jej $L(G)$, je definován:

$$L(G) = \{u;\ u \in T^*, S \Rightarrow^* u\}.$$

x+y

Příklad 2.3.2.
Uvažujme gramatiku

$$= (\{, A, C, 1, 2, 3\}, \{ , , c\}, ,)$$

kde

$$= \cup \cup ,$$

$$= \{ \to A C3, A \to A C, A \to 1\},$$

$$= \{C \to C\},$$

$$= \{1 \to 1, 1C \to c2, 2C \to c2, 23 \to \varepsilon\}.$$

Na první pohled je vidět, že tato gramatika vygeneruje podle pravidel z množinu řetězců $\{ {}^{i}1(C)^{i}3: i \geq 1\}$.

Pravidlo z umožňuje zaměnit řetěz dvou sousedních symbolů C na řetěz C. Pravidla z kontrolují, zda se všechny výskyty symbolu vyskytují před výskyty symbolu C, jenž je pak (prostřednictvím pravidla $1C \to c2$ resp. $2C \to c2$) přepsán na terminál c.

Uvažujeme derivaci:

$$\begin{aligned}
&\Rightarrow A C3\\
&\Rightarrow A C C3\\
&\Rightarrow 1 C C3\\
&\Rightarrow 1 CC3\\
&\Rightarrow 1 CC3\\
&\Rightarrow 1 1CC3\\
&\Rightarrow 1 c2C3\\
&\Rightarrow 1 cc23\\
&\Rightarrow 1 cc.
\end{aligned}$$

a podívejme se, jak se provede ku příkladu čtvrtý krok derivace, tj. $1 C C3 \Rightarrow 1 CC3$. Tuto přímou derivaci jsme provedli pomocí pravidla $C \to C$: našli jsme podslovo ve slovu $1 \underline{C C} 3$, které je rovno levé straně tohoto pravidla (viz podtržená dvojice symbolů), a přepsali (tzn. nahradili) jsme jej pravou stranou pravidla $C \to C$, takže jsme získali slovo $1 CC3$. Nyní uvažujeme jinou derivaci:

$$\Rightarrow\ A\,\flat\,C3$$
$$\Rightarrow\ A\,\flat\,C\,\flat\,C3$$
$$\Rightarrow\ 1\,\flat\,C\,\flat\,C3$$
$$\Rightarrow\ \flat\,1C\,\flat\,C3$$
$$\Rightarrow\ \flat\,c2C3$$

a všimněme si, že další derivační krok již nemohu provést, neboť ve slovu $\flat\,c2\,\flat\,C3$ neexistuje podslovo, jež by bylo rovno levé straně některého pravidla. Ze slova $\flat\,c2\,\flat\,C3$ nelze tudíž derivovat žádné slovo z ( ).

Intuitivně by tedy čtenáři mělo být zřejmé, že

$$(\ ) = \{\ ^{n}\flat\,^{n}c^{n}\colon n \geq 1\};$$

formální důkaz tohoto tvrzení ponecháváme jako cvičení.

Místo „ (přímo) derivuje “ se v literatuře setkáváme i s jinými termíny, např. „ (přímo) generuje “, „ je (přímo) odvozeno z “ apod. Podobně se místo „počáteční symbol“ užívá někdy termínu „axiom“ popř. „startující symbol“. Všechny tyto termíny je proto třeba chápat jako synonyma příslušných pojmů.

Každé slovo, které lze odvodit z počátečního symbolu dané gramatiky, nazýváme větná forma. Pokud větná forma patří do jazyka generovaného touto gramatikou (tzn. že se jedná o větnou formu nad abecedou terminálů), nazýváme ji věta.

Jistě nás po výše uvedené obecné definici generativní gramatiky napadla zcela fundamentální otázka: Lze každý jazyk generovat vhodnou gramatikou? Odpověď zní záporně. Proč? Množina všech slov nad nějakou abecedou je nekonečně spočetná. Jazyk je libovolná podmnožina této množiny všech slov. Z teorie množin víme, že systém všech podmnožin nekonečně spočetné množiny není spočetný. Gramatik je spočetně mnoho, poněvadž každou jsme schopni popsat nějakým slovem symbolů. Odtud tedy vyplývá, že existují jazyky, pro které neexistuje žádná gramatika, jež by je specifikovala.

Gramatiky zavedené v definici 2.3.1 jsou (zejména co do tvaru pravidel) natolik obecné, že pro naše účely (tj. popis lexikálních symbolů a syntaxe daného programovacího jazyka) nemají prakticky žádný význam. Ukazuje se výhodné pravidla těchto gramatik vhodným způsobem omezit a zavést tak několik speciálních typů generativních gramatik. Tvar pravidel se nám tedy stává kritériem pro specifikaci gramatik. K nejvýznamnějším takovýmto klasifikacím patří definice čtyř základních typů gramatik, jak je zavedl Noam Chomsky již v roce 1956.

DEF

Definice 2.3.3.

1. Gramatiku zavedenou v definici 2.3.1 budeme nazývat gramatika typu 0.
2. Buď = (, , ,) gramatika typu 0 a nechť pro každé pravidlo → z platí | |≤| |; jedinou výjimkou může být pravidlo → ε, jehož výskyt však implikuje, že se nevyskytuje na pravé straně žádného pravidla. Pak tuto gramatiku nazýváme gramatika typu 1 popř. kontextová gramatika.
3. Buď = (, , ,) gramatika typu 0, pro kterou platí: → ∈ implikuje ∈ . Pak nazýváme gramatika typu 2 neboli bezkontextová gramatika.
4. Buď = (, , ,) gramatika typu 0 a nechť každé pravidlo z je buď tvaru $A \rightarrow xB$, nebo tvaru $A \rightarrow x$, kde A, $B \in$, $x \in$ *. Pak nazýváme gramatika typu 3 neboli regulární.

Máme-li gramatiku typu i, $i \in \{0, 1, 2, 3\}$, pak jazyk, který tato gramatika generuje, nazýváme jazyk typu i.

Podobně jako tomu bylo v případě gramatik, budeme i jazyky typu 2 často nazývat bezkontextové a jazyky typu 3 regulární.

Velmi snadno lze ukázat (viz např. [2]), že třída jazyků typu i je vlastní podtřída jazyků typu i-1 (tj. každý jazyk typu i je také typu i-1, ale existují jazyky typu i-1, které nelze generovat žádnou gramatikou typu i) pro i = 1, 2, 3. Gramatikami typu 0 a 1 se v tomto textu zabývat nebudeme. O to větší pozornost budeme věnovat regulárním a bezkontextovým gramatikám, které mají pro definici programovacích jazyků zásadní význam.

V tomto textu také zavedeme dva typy automatů: (i) konečný a (ii) zásobníkový. Tyto automaty úzce souvisí s gramatikami typu 2 a 3, o čemž svědčí následující dvě tvrzení, která později dokážeme – viz odst. 2.4 a 2.5 (pojem „jazyk definovaný konečným resp. zásobníkovým automatem“, kterého níže užijeme, bude také zaveden až v těchto odstavcích):

- je bezkontextový jazyk, právě když jej lze definovat vhodným zásobníkovým automatem.
- je regulární jazyk, právě když jej lze definovat vhodným konečným automatem.

Později poznáme ještě řadu dalších důvodů, proč se konečnými a zásobníkovými automaty v tomto textu musíme zabývat. Prozatím stačí poznamenat, že tyto automaty tvoří základ prostředků, pomocí kterých realizujeme některé hlavní části kompilátorů (zejména lexikální a syntaktický analyzátor).

Tento odstavec uzavřeme jednoduchou, ale důležitou definicí:

DEF

Definice 2.3.5.

Dva automaty resp. gramatiky nazýváme ekvivalentní, jestliže definují tentýž jazyk.

2.4. Konečné automaty, regulární výrazy a regulární gramatiky

V titulu této kapitoly jsme uvedli tři základní prostředky pro specifikaci libovolného regulárního jazyka. Postupně si je probereme a ukážeme si jejich vzájemnou těsnou souvislost.

2.4.1 Konečné automaty

Konečné automaty jsou prvním prostředkem pro specifikaci regulárních jazyků, kterými se v naší práci budeme zabývat. Tyto automaty jsou pro konstrukci kompilátorů velmi důležité, neboť tvoří základ tzv. konečných převodníků, pomocí kterých je realizována první fáze procesu kompilace – lexikální analýza.

Neformálně řečeno, konečný automat má konečnou množinu stavů, ve které je specifikován počáteční stav a množina koncových stavů. Pracuje tak, že čte dané slovo nad vstupní abecedou zleva doprava, symbol po symbolu. V prvním kroku je v počátečním stavu, čte nejlevější symbol vstupního slova a přejde do následujícího stavu, ve kterém čte druhý symbol atd., až celé slovo přečte do konce. Automat vstupní slovo akceptuje, jestliže po jeho přečtení skončí v některém koncovém stavu, jinak jej neakceptuje.

Formálně:

DEF

Definice 2.4.1.

Nedeterministický konečný automat A je pětice $A = (Q, \ , \delta, \ , F)$, kde:

(i) Q je abeceda stavů
(ii) je abeceda vstupních symbolů
(iii) δ: $Q \times \ \rightarrow 2^Q$ je přechodová funkce (2^Q označuje množinu všech podmnožin množiny Q)
(iv) $\in Q$ je počáteční stav
(v) $F \subseteq Q$ je množina koncových stavů.

Dvojici (q, x), $q \in Q$, $x \in \ ^*$, nazýváme konfigurace automatu A. Konfigurace $(, x)$, $x \in \ ^*$, se nazývá počáteční konfigurace.

Buď (p, x), $(q, \ )$ dvě konfigurace automatu A. Řekneme, že automat přímo přejde z konfigurace (p, x) do konfigurace $(q, \ )$, symbolicky $(p, x) \vdash (q, \ )$, jestliže $x = \ $, pro nějaké $\in$ a $q \in \delta(p, \ )$. Relaci $\vdash$ nazýváme takt nebo přechod. Pro ≥ 1 píšeme $(p, x) \vdash \ (q, \ )$, jestliže buď $= 1$ a $(p, x) \vdash (q, \ )$, a nebo > 1 a existuje konfigurace $(r, \ )$ taková, že $(p, x) \vdash (r, \ )$ a $(r, \ ) \vdash^{-1} (q, \ )$. Řekneme, že automat přejde z konfigurace (p, x) do

konfigurace (q,), symbolicky (p, x) $\vdash^*$ (q,), jestliže buď (p, x) = (q,), a nebo (p, x) $\vdash$ (q,) pro nějaké ≥ 1.

Jazyk akceptovaný (definovaný) automatem A je označen (A) a definován

(A) = $\{x: x \in$ *, ($\bullet$, x) $\vdash^*$ (r, ε), $r \in F\}$.

x+y

Příklad 2.4.2.

Představme si, že máme za úkol sestrojit nedeterministický konečný automat, který akceptuje každý řetěz nad binární abecedou {0, 1}, který končí symbolem 1. Pak jedním z možných řešení je následující automat:

$$A = (\{p, q\}, \{0, 1\}, \delta, p, \{q\}),$$

kde

$$\delta(p, 0) = \delta(p, 1) = \{p\},$$
$$\delta(p, 1) = \{q\}.$$

Automat A má tedy neustále možnost zůstávat v počátečním stavu p a číst symbol 0 nebo 1. Má-li však přejít do koncového stavu q, pak posledním přečteným symbolem musí být 1.

Tak např. slovo 011 bude automat A akceptovat takto:

$$(p, 011) \vdash (p, 11)$$
$$\vdash (p, 1)$$
$$\vdash (q, \varepsilon).$$

Slovo 010 ovšem do (A) nepatří, neboť buď

$$(p, 010) \vdash (p, 10)$$
$$\vdash (q, 0)$$

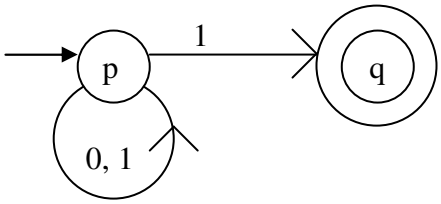
a automat nemůže provést další takt a přitom vstupní slovo nedočetl do konce, a nebo

$$(p, 010) \vdash (p, 10)$$
$$\vdash (p, 0)$$
$$\vdash (p, \varepsilon)$$

V druhém případě sice automat A dočetl vstupní slovo do konce, avšak nepřešel do koncového stavu q.

Zřejmě tedy: (A) = $\{0, 1\}^*1$, což bylo naším cílem.

Konečný automat je velmi často reprezentován pomocí tzv. stavového diagramu. Jedná se o orientovaný graf (viz odst. 2.5.1.1), jehož vrcholy odpovídají stavům automatu, hrany označují možné přechody mezi stavy a jsou ohodnoceny příslušnými vstupními symboly. Vrchol odpovídající počátečnímu stavu je označen šipkou, koncové stavy jsou označeny uzly s dvojitým kroužkem. Tak například automat A z příkladu 2.4.2 lze jednoduše definovat:



Existují ještě další reprezentace konečných automatů (stavové stromy, tabulky apod.). Čtenář nalezne jejich definice např. v [9], [12].

Všimněme si, že pro libovolnou konfiguraci C automatu definovaného v 2.4.1 může obecně existovat několik konfigurací, do kterých může automat z C (přímo) přejít. Tato situace nastala i v případě automatu A z příkladu 2.4.2: ze stavu p mohl automat při čtení symbolu 1 přejít jak do stavu q, tak zůstat ve stavu p. Takovýto nedeterminismus je z praktického hlediska zřejmě krajně nepříjemná vlastnost, kterou se musíme pokusit odstranit.

DEF

Definice 2.4.3.
Buď $A = (Q, \ , \delta, \ , F)$ nedeterministický konečný automat a nechť pro libovolné $\in$, $q \in Q$ existuje nejvýše jeden stav $p \in Q$ tak, že $p \in \delta(q, \ )$. Pak A nazýváme deterministický konečný automat.

Pro libovolnou konfiguraci deterministického konečného automatu existuje tedy nejvýše jedna konfigurace, do které může tento automat přímo přejít. Tím je (výše uvedený) nedostatek nedeterministických konečných automatů odstraněn. Vzniká však otázka, zda lze ke každému nedeterministickému automatu nalézt deterministický konečný automat, který bude přijímat stejný jazyk.

Věta 2.4.4.
Buď A nedeterministický konečný automat. Pak existuje deterministický konečný automat A' tak, že $(A) = (A')$.

Důkaz:
Nechť

$$A = (Q, \ , \delta, \ , F)$$

je libovolný nedeterministický konečný automat. Definujeme deterministický konečný automat

$$A' = (Q', \ , \delta', \ , F')$$

takto:

$$Q' = 2^Q;$$
$$s' = \{s\}$$
$$F' = \{ \in 2^Q : \cap F \neq \emptyset\}.$$

Funkce:

$$\delta' : 2^Q \times \to 2^Q$$

je definována předpisem:

$$\delta'(,) = \bigcup_{q\in} \delta(q,),$$

pro libovolné $\in 2^Q$, $\in$.

Tím je konstrukce automatu A' ukončena. Nyní musíme dokázat, že A' a A jsou ekvivalentní (tzn. $(A) = (A')$).

Pro prázdné slovo je situace jasná:

$\varepsilon \in (A)$, právě když $s \in F$, a tedy právě když $\{s\} = s' \in F'$, což platí, právě když $\varepsilon \in (A')$.

Zbývá dokázat, že pro každé $\notin \varepsilon$, platí:

$$\in (A), \text{ právě když } \in (A').$$

(i) Nechť $= {}_1 \ldots {}_n \in (A'), n \geq 1$. Pak existují ${}_1, \ldots, {}_{n+1} \in 2^Q$ tak, že $s' = {}_1, {}_{n+1} \in F'$, ${}_{i+1} = \delta'({}_i, {}_i)$, $1 \leq i \leq n$. Nejprve ukážeme, že pro každé r, $0 \leq r \leq n$-1, existují stavy $q_{n-r}, q_{n-r+1}, \ldots, q_{n+1}$ tak, že $q_{n+1} \in {}_{n+1}$, $q_{n+1} \in F$, $q_{i+1} \in \delta(q_i, {}_i)$, $q_i \in {}_i$, $n - r \leq i \leq n$. Důkaz bude proveden indukcí podle r.

(1) Základ indukce: $r = 0$. ${}_{n+1} \in F'$ a tedy ${}_{n+1} \cap F \neq \emptyset$. Nechť q_{n+1} je jistý stav z ${}_{n+1} \cap F$. Protože $q_{n+1} \in {}_{n+1} = \delta'({}_n, {}_n)$, musí existovat stav $q_n \in {}_n$ tak, že $q_{n+1} \in \delta(q_n, {}_n)$.

(2) Indukční předpoklad: Nechť tvrzení platí pro nějaké r, $0 \leq r \leq n$-1. Dokážeme toto tvrzení pro $r+1$ (indukční krok).

Z indukčního předpokladu vyplývá, že existuje posloupnost stavů $q_{n-r}, \ldots, q_{n+1}$, které splňují požadované vlastnosti. Protože $q_{n-r} \in {}_{n-r} = \delta'({}_{n-r-1}, {}_{n-r-1}) = \bigcup_{q\in {}_{n-r-1}} \delta(q, {}_{n-r-1})$, existuje $q_{n-r-1} \in {}_{n-r-1}$ takové, že $q_{n-r} \in \delta(q_{n-r-1}, {}_{n-r-1})$. Tím jsme ovšem dokázali tvrzení pro každé r, $1 \leq r \leq n$-1. Uvažujme toto tvrzení pro $r = n$-1. Jinak řečeno, existuje posloupnost $q_1, \ldots, q_{n+1}$ požadovaných vlastností ($q_1 \in {}_1$ a tedy $q_1 \in \{s\}$). Odtud, ${}_1 \ldots {}_n = \in (A)$. Tedy, $(A') \subseteq (A)$.

(ii) Nechť = $_1 \ldots _n \in$ (A). Pak existuje posloupnost stavů $q_1, \ldots, q_{n+1}$ (z Q), pro kterou platí:

(a) $q_1 =$;

(b) $q_{i+1} \in \delta(q_i, \ _i)$, $1 \le i \le n$;

(c) $q_{n+1} \in F$

Buď $_1 =$ ′, $_{i+1} = \delta'(\ _i, \ _i)$, $1 \le i \le n$. Z (a) plyne $q_1 \in$ ′. Z (b) lze indukcí snadno dokázat: $q_j \in \ _j$, $1 \le j \le n+1$. Pak ovšem z (c) dostáváme $_{n+1} \cap F \neq \emptyset$. Odtud $_{n+1} \in F'$ a tedy $_1 \ldots _n =$ $\in$ (A'). Tím jsme dokázali inkluzi $(A) \subseteq$ (A'). Z (i) a (ii) tedy plyne $(A) =$ (A').

Právě jsme tedy dokázali, že každý jazyk definovaný nedeterministickým konečným automatem lze popsat vhodným deterministickým konečným automatem, jehož realizace je v praxi vždy podstatně snazší (jako je tomu konec konců ve všech případech, kdy máme možnost pracovat s deterministickým prostředkem místo s nedeterministickým). Později poznáme, že v případě zásobníkových automatů je situace méně uspokojivá, neboť ne každý nedeterministický zásobníkový automat lze nahradit ekvivalentním deterministickým zásobníkovým automatem (viz odst. 2.5.2).

Všimněme si, že důkaz věty 2.4.4 podává přímo algoritmus transformace nedeterministického konečného automatu na deterministický. Poznamenejme, že takové důkazy (mezi které patří většina důkazů v tomto textu) nazýváme konstruktivní (popř. efektivní).

Dohodněme se, že deterministické konečné automaty budeme obvykle nazývat jako konečné automaty.

Zabývejme se nyní uzávěrovými vlastnostmi třídy jazyků akceptovaných konečnými automaty. Dokážeme, že tato třída je uzavřena vůči sjednocení, zřetězení a iterací.

Věta 2.4.5.
Buď $_1 =$ (A_1), $_2 =$ (A_2), kde A_1, A_2 jsou nějaké konečné automaty. Pak také jazyky

(i) $_1 \cup \ _2$;
(ii) $_1 \ _2$;
(iii) $^*_1$
patří do třídy jazyků akceptovaných konečnými automaty.

Důkaz:
Buď

$$A_1 = (Q_1, \quad , \delta_1, \text{ }_1, F_1),$$
$$A_2 = (Q_2, \quad , \delta_2, \text{ }_2, F_2)$$

konečné automaty. Bez újmy na obecnosti předpokládejme, že $Q_1 \cap Q_2 = \varnothing$ (kdyby Q_1 a Q_2 nebyly disjunktní množiny, pak bychom stavy v jedné z těchto množin přejmenovali) a že pro každé $q \in Q_i$, $\in$ existuje $\delta_i(q, \ ) \in Q_i$ (pokud by pro některé $q \in Q_i$, $\in$ bylo $\delta_i(q, \ )$ nedefinováno, zavedli bychom do Q_i nový stav, řekněme $\#$, a definovali $\delta_i(q, \ ) = \#$; nadto bychom vybrali zcela libovolně jeden symbol z , řekně

(c) pro každé $q \in Q_2$: $\delta(q, \ ) = \delta_2(q, \ )$.

Automat A nejprve simuluje automat A_1 (srov. (a)). Jakmile dosáhne koncového stavu z A_1, může přejít do počátečního stavu s_2 a simulovat činnost automatu A_2.

Buď $x \in \ (A_1)$, $\ \in \ (A_2)$. Pak $(s_1, x\ ) \vdash^* (q, \ )$ v A pro nějaké $q \in F_1$. Jestliže $\ \neq \varepsilon$, pak A provede jeden takt dle (b) a nula nebo více taktů podle (c) tak, že $(q, \ ) \vdash^i (r, \varepsilon)$, pro nějaké $r \in F_2$, $i \geq 1$. Pokud $\ = \varepsilon$, pak $q \in F$, neboť $s_2 \in F_2$. Odtud, $x \ \in \ (A)$.

Nechť $\ \in \ (A)$. Pak $(s_1, \ ) \vdash^* (q, \varepsilon)$, $q \in F$. Předpokládejme nejprve, že $q \in F_2$. Pak $\ = x$ pro nějaké $\ \in \ $ tak, že:

$$(s, x \ ) \vdash^* (p, \ ) \vdash^* (\overline{p}, \ ) \vdash^* (q, \varepsilon)$$

$$\text{v } A, \text{ kde } p \in F_1, \ \overline{p} \in Q_2, \ \overline{p} \in \delta_2(p, \ )$$

Odtud, $x \in \ (A_1)$, $\ \in \ (A_2)$.

Nyní předpokládejme $q \in F_1$. Pak ale $s_2 \in F_2$ a $\varepsilon \in \ (A_2)$. Tedy $\ \in \ (A_1)$. Tím jsme dokázali $\ (A) = \ (A_1) \ (A_2)$.

(iii) Nyní zavedeme nedeterministický konečný automat

$$A = (Q_1 \cup \{q_1\}, \ , \delta, q_1, F_1 \cup \{q_1\})$$

o kterém dokážeme: $\ (A) = (\ (A_1))^*$.

Buďte

(1) q_1 nový stav, tj. $q_1 \notin Q_1$;

(2) (a) pro každé $\ \in \ $, $q \in Q_1 - F_1$: $\delta(q, \ ) = \delta_1(q, \ )$;

(b) pro každé $\ \in \ $, $q \in F_1$: $\delta(q, \ ) = \delta_1(q, \ ) \cup \delta_1(s_1, \ )$;

(c) pro každé $\ \in \ $: $\delta(q_1, \ ) = \delta_1(s_1, \ )$.

Jakmile se automat A dostane do koncového stavu automatu A_1, může pokračovat v simulování automatu A_1 a nebo začít simulovat činnost automatu A_1 z počátečního stavu. Důkaz, že $\ (A) = (\ (A_1))^*$ je analogický jako v části (ii). Poznamenejme, že z $q_1 \in F_1$ plyne $\varepsilon \in \ (A)$.

V části (i), (ii), (iii) byl zaveden nedeterministický konečný automat A. Podle předcházející věty 2.4.4 však k němu můžeme zkonstruovat ekvivalentní deterministický konečný automat, čímž je důkaz věty dokončen.

O třídě jazyků akceptovaných konečnými automaty lze také ukázat, že tvoří booleovu algebru, tzn. že je uzavřená nejen vůči sjednocení (jak jsme výše dokázali), ale též vůči průniku a doplňku. Důkaz tohoto tvrzení

spolu s dalšími výsledky týkajících se uzávěrových vlastností třídy jazyků definované konečnými automaty nalezne čtenář v [2] a [9].

2.4.2. Regulární výrazy

Lexikální symboly daného programovacího jazyka (identifikátory atd.) musí být vytvořeny podle určitých pravidel. Tak např. v případě programovacího jazyka PASCAL musí každý identifikátor začínat písmenem, v řetězci znaků musí být první a poslední symbol apostrof apod. Velmi jednoduchým nástrojem, kterým lze požadované vlastnosti lexikálních symbolů zadat, jsou regulární výrazy.

DEF

Definice 2.4.6.
Buď V abeceda. Regulární množina nad V je definována rekurzivně:

(1) $\emptyset$ je regulární množina nad V;
(2) $\{\varepsilon\}$ je regulární množina nad V;
(3) jestliže $a \in V$, pak $\{a\}$ je regulární množina nad V;
(4) jestliže R_1 a R_2 jsou regulární množiny nad V, pak:
   (a) $R_1 \cup R_2$
   (b) $R_1 R_2$
   (c) R_1^*
   jsou regulární množiny nad V;
(5) žádná jiná regulární množina než vytvořená podle (1)-(4) nad V neexistuje.

DEF

Definice 2.4.7.
Buď V abeceda. Regulární výraz r nad V je definován rekurzivně:

(1) $\emptyset$ je regulární výraz označující regulární množinu $\emptyset$;
(2) ε je regulární výraz označující regulární množinu $\{\varepsilon\}$;
(3) jestliže $a \in V$, pak a je regulární výraz označující regulární množinu $\{a\}$;
(4) jestliže r_1 a r_2 jsou regulární výrazy označující regulární množiny R_1 a R_2, pak:
   (i) $(r_1 + r_2)$ je regulární výraz označující regulární množinu $R_1 \cup R_2$;
   (ii) $(r_1 r_2)$ je výraz označující regulární množinu $R_1 R_2$;
   (iii) r_1^* je regulární výraz označující regulární množinu R_1^*;
(5) žádné jiné regulární výrazy, než vytvořené podle (1)-(4) nad abecedou V neexistují.

Označuje-li regulární výraz r regulární množinu R, pak říkáme, že R je reprezentován regulárním výrazem r.

Je jasné, že pro každou regulární množinu lze nalézt vhodný regulární výraz, který ji reprezentuje, a také naopak platí, že každý regulární výraz reprezentuje nějakou regulární množinu.

Je ovšem též zřejmé, že několik regulárních výrazů může reprezentovat stejnou regulární množinu (lze dokonce velmi snadno ukázat, že pro libovolnou regulární množinu existuje nekonečně mnoho regulárních výrazů, které ji reprezentují).

Ku příkladu regulární výraz ( )* nad abecedou { } označuje množinu { }*, která však může být reprezentována i regulárními výrazy ( )*(∅) či ( )*( )*. O takovýchto regulárních výrazech (které reprezentují identickou regulární množinu) říkáme, že jsou si rovny.

Pro zpřehlednění zápisu zpravidla vynecháváme zbytečné závorky; např. místo regulárního výrazu ((( ))) nad abecedou { , } budeme jednoduše psát apod.

Již jsme se v úvodu odstavce zmínili, že pomocí regulárních výrazů lze většinou poměrně jednoduchým způsobem popsat lexikální symboly daného programovacího jazyka. Ilustrujme si to příkladem:

Příklad 2.4.8

Pravidla pro vytvoření identifikátorů v PASCALu lze zadat pomocí regulárních výrazů takto:

⟨PISMENO⟩ = A + B + C + D + E + F + F + G + H + I + J + K + L + M +
N + O + P + Q + R + S + T + U + V + W + X + Y + Z;

⟨CISLICE⟩ = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9;

⟨IDENTIFIKATOR⟩ = ⟨PISMENO⟩ (⟨CISLICE⟩ + ⟨PISMENO⟩)*

Podle definice, každý regulární výraz označuje jazyk (tj. množinu slov nad nějakou abecedou), který nazýváme regulární množina. Ukážeme si, že třída regulárních množin je identická s třídou jazyků akceptovaných konečnými automaty.

Lemma 2.4.9.

Buď abeceda. Pak jazyk:

(i) ∅;
(ii) {ε};
(iii) { }, ∈ ;

lze akceptovat vhodným konečným automatem.

Důkaz:

(i) Buď A libovolný automat s prázdnou množinou koncových stavů. Pak $(A) = \emptyset$.

(ii) Buď $A = (\{\ \}, , \delta, , \{\ \})$ a pro každé ∈ je $\delta(\ ,\ )$ nedefinováno. Pak $(A) = \{\varepsilon\}$.

(iii) Buď ∈ . Definujme $A = (\{r,\ \}, , \delta, , \{r\})$, $\delta(\ ,\ ) = r$ a pro všechny ostatní případy je funkce δ nedefinovaná. Pak $(A) = \{\ \}$.

Věta 2.4.10.
Buď = regulární množina. Pak existuje konečný automat A tak, že = (A).

Důkaz: Plyne z Lemma 2.4.9 a Věty 2.4.5.

Věta 2.4.11.
Každý jazyk akceptovaný konečným automatem je regulární množina.

Důkaz:
Buď

$$A = (Q, \ , \delta, q_1, F)$$

libovolný konečný automat, = (A). Nechť $Q = \{q_1, \ldots, q_n)$. Pro každé i, j, $1 \le i, j \le n$, definujeme:

$$R_{ij} = \{ \ \in \ ^*: \delta(q_i, \ ) = q_j\}.$$

Je zřejmé, že (A) = $\bigcup_{q_j \in F} R_{ij}$. Povšimněme si, že je vyjádřeno pomocí konečného počtu sjednocení (což je regulární operace) R_{ij}. K tomu, abychom dokázali, že je regulární množina, bude tedy jistě postačující dokázat, že R_{ij} je regulární množina.

Buď R_{ij} množina slov převádějící automat ze stavu q_i do q_j bez meziprůchodu jakýmkoli stavem q_{k}, pro který platí $k >$. Dokážeme, že pro každé i, j, , $1 \le i, j,$ $\le n$, je R_{ij} regulární množina. Tato skutečnost nám pak bude implikovat, že i je regulární množina, neboť zřejmě = $R_{ij} = R_{ij}^n$.

Důkaz provedeme indukcí podle .

Buď = 0. Pro libovolné i, j je zřejmé, že $R_{ij}^0 \subseteq V \cup \{\varepsilon\}$, což je regulární množina.

Nechť R_{ij} je regulární množina pro všechna i, j a nějaké , kde $0 \le < n$. Dokážeme, že i R_{ij}^{+1} je regulární množina pro libovolné i, j, jestliže slovo převádí automat z q_i do q_j bez průchodu stavem q_{k}, $k >$ + 1, pak buď:

(i) nedojde k meziprůchodu stavem q_{+1} a $\in R_{ij}$.

a nebo

(ii) dojde k $k \ge$ l, meziprůchodům stavem q_{+1}. Pak lze slovo vyjádřit ve tvaru = $u_1 u_2 \ldots u_{k-1}$, kde k dělících bodů mezi podslovy $u_1, \ldots, u_{k-1}$, odpovídají jednotlivým meziprůchodům stavem q_{+1}. Pak je

$u \in R_{i, +1}$,

$\in R_{+1, j'}$,

$_1 \in$ $_{+1,\ +1}$ pro každé $l \le$ - l a tedy

$\in$ $_{i,\ +1}(\ _{+1;\ +1})^*\ _{+1,j'}$

Z (i) a (ii):

$$_{ij}^{+1} = _{ij} \cup\ _{i,\ +1}(\ _{+1,\ +1})^*\ _{+1,j'}$$

Na pravé straně této rovnosti se vyskytují pouze – dle indukčního předpokladu – regulární množiny, se kterými jsou prováděny regulární operace. Proto je $_{ij}^{+1}$ také regulární množina, což jsme měli dokázat.

Z předcházejících dvou vět získáváme toto důležité tvrzení:

Důsledek 2.4.12

Jazyk je akceptovatelný konečným automatem, právě když jej lze reprezentovat regulárním výrazem.

2.4.3. Regulární gramatiky

Tyto gramatiky jsme zavedli již v kapitole 2.3. Nyní jim budeme věnovat poněkud hlubší pozornost.

Nejprve si ukážeme jejich úzkou souvislost s konečnými automaty.

Věta 2.4.13.

Pro každou regulární gramatiku existuje nedeterministický automat A tak, že ( ) = (A) a naopak.

Důkaz:

(i) Buď = (, , ,) regulární gramatika. Definujme nedeterministický konečný automat A = ($\cup \{F\}$, , δ, , $\{F\}$), kde:

(1) F – nový stav;

(2) (a) pro každé $X, Y \in$, $\in$: jestliže $X \to Y \in$ pak $Y \in \delta(X,\ )$

(b) pro každé $X \in$, $\in$: jestliže $X \to \in$ pak $F \in \delta(X,\ )$.

Vysvětleme si hlavní myšlenku důkazu, že ( ) = (A). Uvažujme slovo $_1 \ldots\ _n \in$ ( ), $_i \in$, $1 \le i \le n$, $n \ge 1$. Toto slovo je v odvozeno derivací tohoto tvaru:

$$\begin{aligned} X_0 &\Rightarrow\ _1X_1 \\ &\Rightarrow\ _1\ _2X_2 \\ &\quad\ \vdots \\ &\Rightarrow\ _1\ _2 \ldots\ _{n-1}X_{n-1} \end{aligned}$$

$\Rightarrow$ 1 2 ⋯ n-1 n,

kde $X_0 =$, $X_i \in$, $0 \le i \le n$ - 1.

Podle definice automatu A proběhne akceptování slova 1… n takto:

$$(X_0, \ 1\ 2\ 3\ldots\ n) \vdash (X_1, \ 2\ 3\ldots\ n) \vdash (X_2, \ 3\ldots\ n) \vdash \cdots \vdash (X_{n-1}, \ n) \vdash (F, \varepsilon)$$

Mělo by být tedy zřejmé, že jazyk akceptovaný nedeterministickým konečným automatem A je roven jazyku, který generuje gramatika . Formální důkaz tohoto tvrzení, tj.

$$(A) = \ (\ )$$

ponecháváme jako cvičení – čtenář jej může nalézt např. [9].

(ii) Nyní dokážeme naše tvrzení naopak. Buď $A = (Q, \ , \delta, , F)$ nedeterministický konečný automat. Definujeme regulární gramatiku $= (Q, \ , ,)$, kde obsahuje tato pravidla:

(1) pro každé $p, q \in Q, \ \in$:
jestliže $q \in \delta(p, \ )$, pak $p \to \ q \in$

(2) pro každé $p \in Q, q \in F, \ \in$:
jestliže $q \in \delta(p, \ )$, pak $p \to \ \in$

Formální důkaz, že

$$(A) = \ (\ )$$

je natolik jednoduchý, že jej opět ponecháváme čtenáři jako cvičení.

Dospíváme k tvrzení, které má pro regulární jazyk zásadní význam. Tento výsledek plyne přímo z příslušných definic, věty 2.4.4, důsledku 2.4.12 a věty 2.4.13:

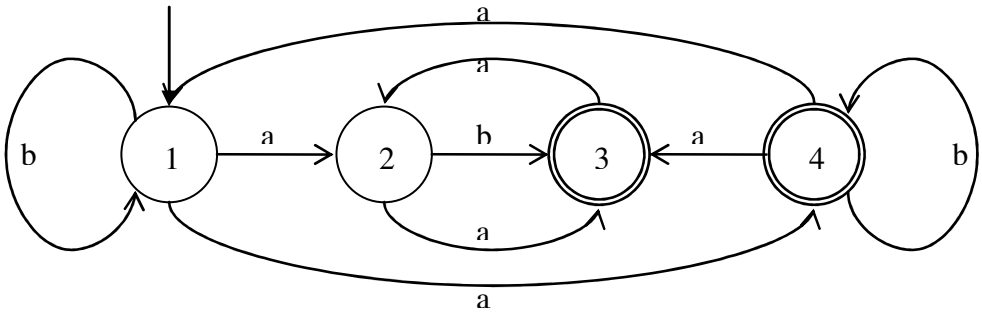
Důsledek 2.4.14.

Následující tvrzení jsou ekvivalentní:

(i) je regulární množina;
(ii) je jazyk reprezentovaný regulárním výrazem;
(iii) $=$ (A_1) pro nějaký konečný automat A_1;
(iv) $=$ (A_2) pro nějaký nedeterministický konečný automat A_2;
(v) je regulární jazyk, tj. $=$ ( ) pro nějakou regulární gramatiku .

2.4.4. Cvičení

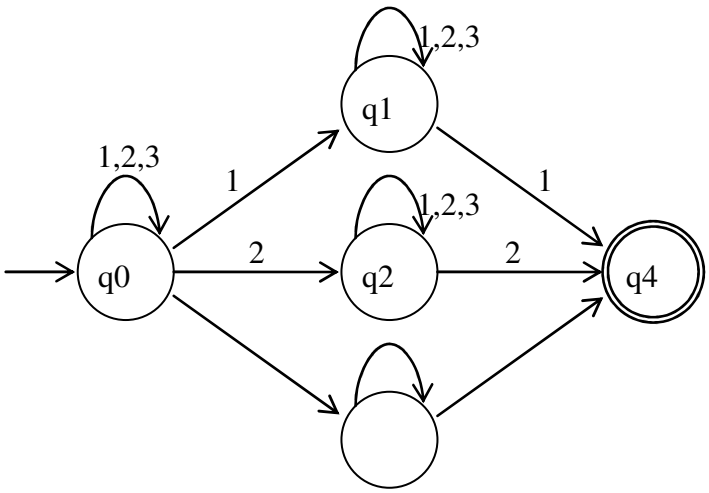
1. Mějme konečný automat A:



$(A) = ?$

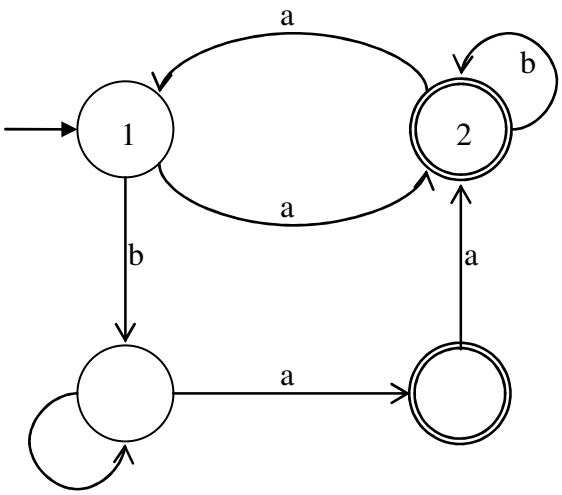
2. Buď $\subseteq \{ , \}^*$ jazyk, který obsahuje všechna slova, ve kterých se vyskytuje lichý počet symbolů , sudý počet symbolů (např. $\in$, ale $\notin$). Sestrojte konečný automat A tak, že $=$ (A).

3. Mějme nedeterministický konečný automat A:



Čemu se rovná (A)? Sestrojte (podle konstrukce uvedené u důkazu věty 2.4.4) deterministický konečný automat A' tak, že $(A) =$ (A').

4. Buď



konečný automat A. Napiš regulární výraz, který reprezentuje (A).

5. Sestrojte konečný automat, který akceptuje jazyk reprezentovaný regulárním výrazem $((\quad)^* + \quad)^*$.

6. Proveďte formální důkaz věty 2.4.13.

2.5. Bezkontextové gramatiky a zásobníkové automaty

Syntaxe programovacích jazyků se nejčastěji definuje pomocí bezkontextových gramatik. Tyto gramatiky úzce souvisí s tzv. zásobníkovými automaty, které tvoří teoretický model syntaktických analyzátorů. Platí totiž, že daný jazyk je bezkontextový, právě když jej lze akceptovat vhodným zásobníkovým automatem.

V této kapitole si probereme nejdůležitější vlastnosti bezkontextových gramatik a uvedeme si základní metody, pomocí kterých lze od libovolné bezkontextové gramatiky přejít k ekvivalentnímu zásobníkovému automatu a naopak.

2.5.1. Bezkontextové gramatiky

Zabývejme se nejprve vlastnostmi bezkontextových gramatik a jazyků.

2.5.1.1. Nejednoznačné bezkontextové gramatiky

Nejednoznačnost dané bezkontextové gramatiky způsobuje při syntaktické analýze značné praktické potíže. Tato vlastnost totiž umožňuje odvodit určité věty generovaného jazyka několika různými způsoby. Abychom mohli o struktuře derivace v dané bezkontextové gramatice nějakým jednoduchým způsobem uvažovat, zavedeme si dva důležité pojmy – tzv. derivační strom a kanonickou derivaci.

Derivační strom je velmi názorný prostředek pro zobrazení struktury derivace v bezkontextové gramatice. Než jej budeme definovat, připomeňme si (alespoň informativně) několik pojmů z teorie grafů, které jsou pro nás důležité.

(Orientovaným) grafem budeme nazývat dvojici $(V, \ )$, kde V je konečná, neprázdná množina (tzv. uzlů) a $\subseteq V \times V$ je množina tzv. hran. O hraně $(\ , b) \in$ říkáme, že vychází z uzlu a vchází do uzlu b, přičemž b nazýváme přímý následovník uzlu . Posloupnost hran $(\ _1, b_1)$, $(\ _2, b_2)$, …,$(\ _n, b_n)$ se nazývá cesta délky n z $\ _1$ do b_n, jestliže $b_i = \ _{i+1}$, $1 \leq i \leq n$ -1, přičemž b_n nazýváme následovník uzlu $\ _1$. Strom je graf, který splňuje tyto dvě podmínky:

(a) existuje právě jeden uzel (tzv. kořen), do kterého nevstupuje žádná hrana,
(b) pro každý uzel b různý od kořene existuje právě jedna cesta z kořene do b.

Uzly stromu, z kterých nevychází žádná hrana, nazýváme listy.

DEF

Definice 2.5.1.

Buď = (, ,) bezkontextová gramatika. Strom je derivační strom v , jestliže:

(1) každý uzel je ohodnocen symbolem z ∪ ,
(2) kořen je ohodnocen symbolem ,
(3) jestliže uzel má nejméně jednoho následovníka, pak je ohodnocen symbolem z ,
(4) jestliže $_1$, $_2$, …, jsou přímí následovníci uzlu , jenž je ohodnocen symbolem A, v pořadí zleva doprava s ohodnocením B_1, B_2, …, B , pak $A \to B_1B_2\ldots B \in$

Ilustrujme si nyní pojem derivačního stromu jednoduchým příkladem:

x+y

Příklad 2.5.2.
Uvažujme bezkontextovou gramatiku

= ({ , F}, { , (,), +, *}, ,)

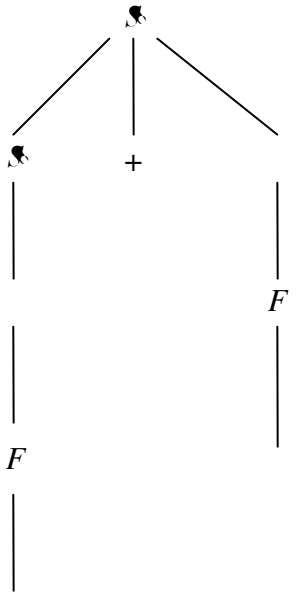
kde

$=\{ \to$,
$\to$ + ,
$\to F$,
$\to F$ * ,
$F \to$,
$F \to ($) }.

Vytvořme derivační strom pro větu:
+

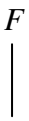
které odpovídá derivace:
$\Rightarrow$ +
$\Rightarrow$ +
$\Rightarrow F$ +
$\Rightarrow F + F$
$\Rightarrow F$ +
$\Rightarrow$ +

Hledaný derivační strom má tvar (pro jednoduchost budeme předpokládat, že hrany jsou v derivačním stromu orientovány směrem dolů a uzly jsou označeny přímo svým ohodnocením):

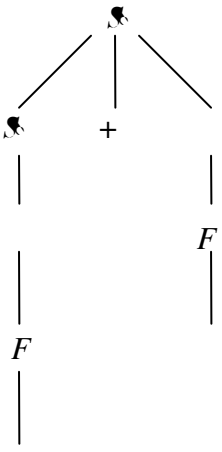


Když budeme číst ohodnocení listů derivačního stromu zleva doprava, pak získáme větnou formu, kterou nazýváme výsledek (derivačního stromu). Ve výše uvedeném příkladě to byla věta + .

Podstrom daného derivačního stromu je uzel tohoto stromu spolu se všemi jeho následovníky, s jejich ohodnocením a s hranami, které je spojují. Ku příkladu:

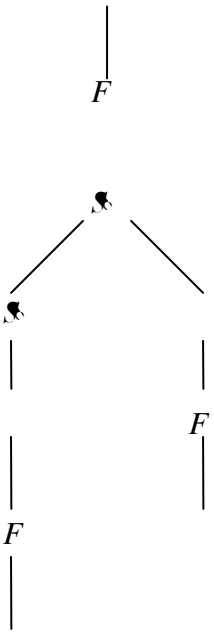


a



jsou podstromy derivačního stromu z příkladu 2.5.2,

kdežto



či

jimi nejsou.

Nyní se budeme zabývat tzv. kanonickými derivacemi, které získáme jistým omezením odvozování slov v bezkontextových gramatikách. Rozlišujeme dva typy kanonických derivací: levé a pravé.

DEF

Definice 2.5.3.

Derivaci nazveme levou (resp. pravou), jestliže v každém jejím kroku nahrazujeme nejlevější (resp. nejpravější) neterminál stávající větné formy. Jestliže tedy

$$x_1 \Rightarrow x_2 \Rightarrow \ldots \Rightarrow x_n$$

je levá (resp. pravá) derivace v gramatice = (, ,), pak pro $1 \le i < n$ lze psát x_i jako $_iA_i\ _i$ (resp. $_iA_i\ _i$), kde $_i \in$ *, $_i \in (\ \cup\ )^*$, $A_i \rightarrow\ _i \in$ $x_{i+1} =\ _i\ _i\ _i$ (resp. $x_{i+1} =\ _i\ _i\ _i$). Všimněme si, že i samotné odpovídá požadovanému výchozímu tvaru větné formy.

Mezi kanonickými derivacemi a derivačními stromy existuje těsná souvislost:
V dané bezkontextové gramatice existuje pro každou větu $x \in$ ( ) levá derivace. Každé levé derivaci věty x odpovídá v jediný derivační strom, jehož je x výsledkem, a naopak (tj. každému derivačnímu stromu odpovídá v jediná levá derivace). Analogická tvrzení platí i pro pravou derivaci; příslušné důkazy nalezne čtenář např. v [2].

Nyní se vraťme k úvahám o nejednoznačnosti bezkontextových gramatik. Výše jsme uvedli, že pro každou větu existuje v dané bezkontextové gramatice (nejméně jedna) levá derivace resp. derivační strom. Na druhé straně ovšem může být daná věta výsledkem několika různých derivačních stromů, tj. lze ji odvodit několika různými levými

derivacemi. A právě takovýto jev činí danou bezkontextovou gramatiku nejednoznačnou.

DEF

Definice 2.5.4.

Bezkontextová gramatika se nazývá nejednoznačná, existuje-li věta $x \in$ ( ), která je výsledkem nejméně dvou různých derivačních stromů. Jinak se nazývá jednoznačná.

x+y

Příklad 2.5.5.

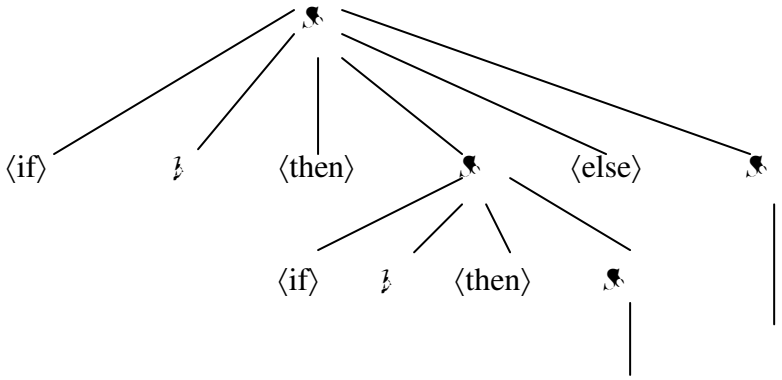
Uvažujme bezkontextovou gramatiku = ({ }, ,)
kde

= { , , ⟨if⟩, ⟨then⟩, ⟨else⟩},
= { → ⟨if⟩ ⟨then⟩ ⟨else⟩ ;
→ ⟨if⟩ ⟨then⟩ ;
→ }

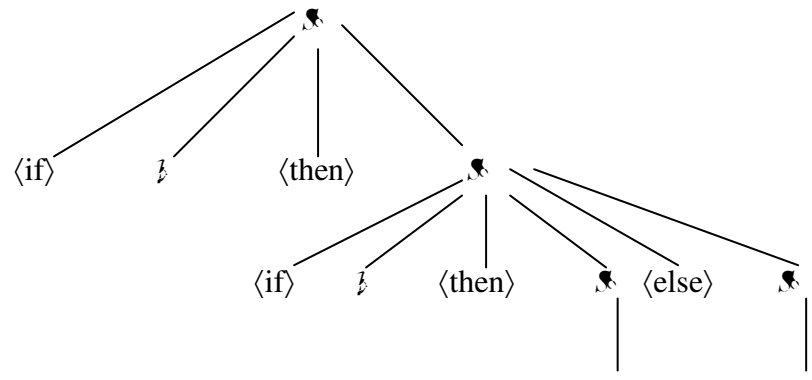
a větu

⟨if⟩ ⟨then⟩ ⟨if⟩ ⟨then⟩ ⟨else⟩

z ( ). Tato věta je výsledkem derivačního stromu



ale i



je proto nejednoznačná gramatika.

Definice 2.5.4 hovoří o nejednoznačnosti bezkontextové gramatiky, nikoliv jazyka. Není totiž vyloučené, že pro danou nejednoznačnou gramatiku existuje ekvivalentní (tj. stejný jazyk generující) bezkontextová gramatika, která je jednoznačná. Ilustrujeme si to opět příkladem:

x+y

Příklad 2.5.6.

Sestrojte bezkontextovou gramatiku , která je jednoznačná a je ekvivalentní s gramatikou z příkladu 2.5.5. Buď

= ({*A*, *B*}, , , *A*),
je stejná jako v ,
= { $A \to$ ⟨if⟩ ⟨then⟩ *A*,
$A \to$ ⟨if⟩ ⟨then⟩ *B* ⟨else⟩ *A*,
$A \to$,
$B \to$ ⟨if⟩ ⟨then⟩ *B* ⟨else⟩ *B*,
$B \to$ }.

V ( ) již neexistuje věta, jež by byla výsledkem více než jednoho derivačního stromu, a proto je jednoznačná. Přitom je zřejmé, že ( ) = ( ).

Vzniká proto otázka: existuje vůbec nejednoznačný bezkontextový jazyk, tj. jazyk, který nelze generovat žádnou jednoznačnou bezkontextovou gramatikou? Žel, je známa existence celé řady takovýchto jazyků; k těm nejznámějším patří:

= { $^{i}\,^{j}c$: $i = j$ nebo $j =$ }.

2.5.1.2 Základní transformace a normální tvary bezkontextových gramatik

Řada úvah o bezkontextových gramatikách se podstatným způsobem zjednoduší, když nalezneme transformace, které libovolnou bezkontextovou gramatiku převedou na ekvivalentní bezkontextovou gramatikou, která splňuje některé potřebné vlastnosti. Může se jednat ku příkladu o eliminaci tzv. zbytečných symbolů, ε-pravidel nebo o zjednodušení tvaru pravidel. Takovéto základní transformace probereme v tomto odstavci.

DEF

Definice 2.5.7.

Buď = (, , ,) bezkontextová gramatika. Neterminál $X \in$ se nazývá zbytečný, jestliže v neexistuje derivace

$$\Rightarrow^{*} \quad X \quad \Rightarrow^{*} \quad x \,, \quad ,x, \quad \in \quad ^{*}.$$

Bezkontextová gramatika taková, že ( ) $\neq \varnothing$, se nazývá redukovaná, jestliže neobsahuje žádný zbytečný neterminál.

Jinak řečeno, zbytečný neterminál se nevyskytuje v žádném odvození terminálního slova z S.

Protože v redukované gramatice je možné z každého neterminálu (samozřejmě včetně S) odvodit terminální slovo, je jazyk touto gramatikou generovaný nutně neprázdný. To je také důvod, proč v definici 2.5.7 předpokládáme $L(G) \neq \emptyset$.

Věta 2.5.8.

Buď $G = (N, T, P, S)$ bezkontextová gramatika, $L(G) \neq \emptyset$. Pak existuje redukovaná bezkontextová gramatika G' taková, že $L(G) = L(G')$.

Důkaz. (algoritmus redukce)
Nechť

$$G = (N, T, P, S)$$

je libovolná bezkontextová gramatika. Nejprve sestrojme množinu symbolů

$$V = \{X \in N \cup T;\ X \Rightarrow^* w \text{ pro nějaké } w \in T^*\}.$$

Položíme

$$V_0 = T$$

a postupně vytváříme množiny

$$V_1, V_2, \ldots$$

kde

$$V_{i+1} = V_i \cup \{X \in N \cup T;\ X \to w \text{ pro nějaké } w \in V_i^*\}.$$

Protože pro každé $i = 0, 1, \ldots$ jistě platí $V_i \subseteq N \cup T$, musí existovat celé číslo $n \geq 0$ tak, že

$$V_n = V_{n+1}$$

a stačí položit

$$V = V_n$$

Tato množina V nám umožňuje zjistit, zda skutečně $L(G) \neq \emptyset$, neboť

$$L(G) \neq \emptyset, \text{ jestliže } S \in V.$$

Jestliže tedy $S \in V$, eliminujme z P všechna pravidla, která na levé nebo pravé straně obsahují symbol z $N - V$, a z množiny T všechny symboly z $T - V$. Takto získáme novou množinu neterminálů N_1 a množinu pravidel P_1. Definujme gramatiku

$$_1 = (\ _1,\ \ ,\ \ ,\ \ ),$$

ve které lze z libovolného neterminálu odvodit nějaké slovo nad , tj. pro každé $x \in$ * tak, že $X \Rightarrow^* x$.

Nyní sestrojme množinu , pomocí které budeme z gramatiky $_1$ eliminovat všechny neterminály, které se nemohou vyskytnout v žádné větné formě (odvozené z). Množinu definujeme

$$= \{X \in \ _1;\ \ \Rightarrow^*\ \ X \quad \text{pro nějaké} \quad \in (\ _1 \cup\ )^*\}.$$

Množinu sestrojíme jednoduše tak, že do ní zavedeme symbol a postupně do ní budeme přidávat všechny neterminály, jež se vyskytují na pravé straně nějakého pravidla $A \rightarrow$, kde A je již ve . Konstrukci ukončíme, jakmile do ní již nelze přidat žádný nový neterminál. Nyní sestrojme novou gramatiku

$$_2 = (\ _2,\ \ ,\ \ ,\ \ )$$

tak, že z $_1$ eliminujeme všechny neterminály z $_1 -$, tj. $_2 =$ $\cap$ $_1$ a z eliminujeme všechna pravidla, na jejichž levé nebo pravé straně se vyskytuje nějaký neterminál nepatřící do , čímž získáme .

Z první části důkazu již víme, že z každého neterminálu z $_2$ lze odvodit nějaké slovo nad . Pro gramatiku $_2$ však nadto platí, že pro libovolný neterminál z $_2$ existuje slovo odvoditelné z , ve kterém se tento neterminál vyskytuje. Jinak řečeno, $_2$ je redukovaná gramatika. Nadto se snadno dokáže, že

$$(\ ) = (\ _1) = (\ _2);$$

Formální důkaz tohoto tvrzení nalezneme v [12].
Ilustrujme si použití algoritmu redukce konkrétním příkladem.

x+y

Příklad 2.5.9
Buď

$$= (\ ,\ \{\ \},\ \ ,\ \ ),$$
$$= \{A,\ B,\ \},$$
$$= \begin{Bmatrix} \rightarrow\ , \\ \rightarrow A, \\ A \rightarrow AB, \\ B \rightarrow \end{Bmatrix}$$

bezkontextová gramatika. Naším úkolem je sestrojit k ní ekvivalentní gramatiku redukovanou. Tuto konstrukci provedeme přesně podle důkazu věty 2.5.8.
Nejprve sestrojme množinu :

$$V_0 = \{\ \},$$
$$V_1 = \{\ ,\ ,\ B\}$$

dále však

$$V_2 = V_1$$

a tedy

$$V\ = V_1 .$$

Protože

$$\ \in V\ ,$$

platí

$$(\ ) \neq \varnothing$$

a můžeme sestrojit gramatiku

$$_1 = (\{\ ,\ B\},\ \{\ \},\ \{\ \rightarrow\ \ ,\ B \rightarrow\ \ \},\ ).$$

Podle druhé části důkazu věty 2.5.8 sestrojíme množinu

$$= \{\ \}$$

a tedy redukovanou bezkontextovou gramatiku

$$_2 = (\{\ \},\ \{\ \},\ \{\ \rightarrow\ \ \},\ ),$$

která je ekvivalentní s .

Často bývá velmi užitečné, eliminovat z množiny pravidel dané bezkontextové gramatiky ε-pravidla, tj. pravidla, jejichž pravá strana je tvořena prázdným slovem. Pokud ovšem prázdné slovo je obsaženo v generovaném jazyku, je samozřejmé, že tento jazyk musí být generován gramatikou, která nejméně jedno ε-pravidlo obsahuje. To nás přivádí k následující definici:

DEF

Definice 2.5.10.
Bezkontextová gramatika = (, , ,) se nazývá bez ε-pravidel, buď jestliže

(i) neobsahuje žádné pravidlo tvaru $X \rightarrow \varepsilon,\ X \in$,

a nebo

(ii) obsahuje jediné ε-pravidlo $S \to \varepsilon$ a S se nevyskytuje na pravé straně žádného z pravidel v

Věta 2.5.11.
Ke každé bezkontextové gramatice existuje ekvivalentní bezkontextová gramatika $_1$ bez ε-pravidel.

Důkaz (algoritmus transformace na gramatiku bez ε-pravidel).
Buď

$$= (\ , \ , \ , S)$$

libovolná bezkontextová gramatika.

Nejprve sestrojíme množinu

$$_\varepsilon = \{X;\ X \in \ , \ X \Rightarrow^* \varepsilon\}.$$

Tuto množinu sestrojíme analogicky jako množinu V v důkazu předcházející věty 2.5.8:

Do $_\varepsilon$ zahrneme všechny neterminály X takové, že $X \to \varepsilon \in$ potom prohlížíme množinu a přidáváme do množiny $_\varepsilon$ všechny ty neterminály X, pro které existuje pravidlo $X \to x$, $x \in \ ^*_\varepsilon$. Tuto proceduru opakujeme tak dlouho, až do $_\varepsilon$ nemůžeme přidat žádny nový neterminál. Protože je konečná, získáváme množinu $_\varepsilon$ po konečném počtu opakovaní této jednoduché procedury.

V dalším kroku sestrojíme novou množinu pravidel a to takto:

(a) jestliže

$$A \to x_0 B_1 x_1 \ldots B \ x \ \in$$
$$B_i \in \ _\varepsilon, 1 \le i \le \ , \ \ge 0$$

a libovolný symbol z x_i
$(1 \le i \le \ )$ nepatří do $_\varepsilon$,

pak do zavedeme všechna pravidla tvaru:

$$A \to x_0 X_1 x_1 \ldots X \ x \ ,$$
$$X_i \in \{B_i, \varepsilon\}, \ x_0 X_1 x_1 \ldots X \ x \ \neq \varepsilon;$$

(b) jestliže $S \in \ _\varepsilon$

pak $\{S_1 \to S, \ S_1 \to \varepsilon\} \subseteq$

a definujme $_1 = \ \cup \{S_1\}$,

kde S_1 je nový počáteční symbol;
jinak $P_1 = P$, $S_1 = S$.

Nakonec položíme

$$G_1 = (N_1, T, P_1, S_1).$$

Bezkontextová gramatika G_1 je již zřejmě bez ε-pravidel a poměrně snadno se dokáže (viz [12]), že

$$L(G) = L(G_1).$$

x+y

Příklad 2.5.12.
Uvažujme bezkontextovou gramatiku

$$G = (\{S\}, \{0, 1\}, \{S \to 0S1S, S \to \varepsilon\}, S)$$

a sestrojme k ní dle důkazu věty 2.5.11 ekvivalentní bezkontextovou gramatiku

$$G_1 = (N_1, \{0, 1\}, P_1, S_1)$$

bez ε-pravidel.
Protože

$$P_\varepsilon = \{S\}$$

zavedeme do P_1 pravidla:

$$S \to 0S1S$$
$$S \to 0S1$$
$$S \to 01S$$
$$S \to 01$$

a poněvadž $S \in P_\varepsilon$:

$$S_1 \to S$$
$$S_1 \to \varepsilon.$$

Pro ilustraci uveďme derivaci 0011 v gramatice G a jemu odpovídající odvození v gramatice G_1:

:

$$\begin{aligned} S &\Rightarrow 0S1S \\ &\Rightarrow 00S1S1S \\ &\Rightarrow 001S1S \\ &\Rightarrow 0011S \\ &\Rightarrow 0011 \end{aligned}$$

$_1$:

$$\begin{aligned} S_1 &\Rightarrow S \Rightarrow 0S1 \\ &\Rightarrow 0011. \end{aligned}$$

Další zjednodušení úvah o bezkontextových gramatikách přináší vyloučení derivací tvaru $A \Rightarrow^+ A$, kde A je neterminál.

DEF

Definice 2.5.13.

Bezkontextová gramatika = (, , ,) se nazývá bez cyklů, jestliže v ní není možná derivace tvaru: $A \Rightarrow^+ A$, $A \in$.

Věta 2.5.14.

Každý bezkontextový jazyk lze generovat vhodnou bezkontextovou gramatikou bez cyklů. (Důkaz tohoto tvrzení nalezneme např. v [2].)

DEF

Definice 2.5.15.

Redukovaná bezkontextová gramatika bez cyklů a bez ε-pravidel se nazývá vlastní.

Věta 2.5.16.

Každý bezkontextový jazyk lze generovat vhodnou vlastní bezkontextovou gramatikou.

Důkaz. Plyne z důkazů předcházejících vět tohoto odstavce.

Nyní se zabývejme transformacemi, které v bezkontextových gramatikách zjednodušují tvar pravidel, přesněji řečeno, libovolnou bezkontextovou gramatiku převádějí na gramatiku, ve které je každé pravidlo určitého předepsaného, tzv. normálního tvaru. Proto se také často hovoří o normálních formách bezkontextových gramatik. My se zde budeme zabývat dvěma nejznámějšími formami. Je to tzv. Chomského normální forma a Greibachové normální forma.

DEF

Definice 2.5.17.

Bezkontextová gramatika = (, , ,) je v Chomského normální formě, jestliže každé pravidlo z je jednoho z následujících tvarů:

(1) $A \to BC$,
(2) $A \to$,

(3) $\to \varepsilon$, jestliže $\varepsilon \in$ ( ), přičemž se nevyskytuje na pravé straně libovolného pravidla,
kde $A, B, C \in$, $\in$.

Věta 2.5.18.
Libovolný bezkontextový jazyk lze generovat vhodnou bezkontextovou gramatikou v Chomského normální formě.

Důkaz.
Buď bezkontextový jazyk a nechť

$$= (, , ,)$$

je vlastní bezkontextová gramatika, která jej generuje, tj. = ( ).

Sestrojme bezkontextovou gramatiku

$$' = (', , ,),$$

jenž je v Chomského normální formě a pro kterou platí

$$( ) = (').$$

Gramatiku ′ sestrojíme takto:

(1) jestliže $A \to x \in , x \in \cup ^2$,
pak $A \to x \in$;
(2) jestliže $\to \varepsilon \in$
pak $\to \varepsilon \in$;
(3) pro každé pravidlo z P tvaru
$A \to X_1 \ldots X$, > 2
zavedeme do pravidla

$$A \to X_1' \langle X_2 \ldots X \rangle$$
$$\langle X_2 \ldots X \rangle \to X_2' \langle X_3 \ldots X \rangle$$
$$\vdots$$
$$\langle X_{-2} X_{-1} X \rangle \to X'_{-2} \langle X_{-1} X \rangle$$
$$\langle X_{-1} X \rangle \to X'_{-1} X'$$

kde $X_i' = X_i$, jestliže $X_i \in$, X_i' je nový neterminál, jestliže $X_i \in$; $\langle X_i \ldots X \rangle$ je nový neterminál;

(4) jestliže $A \to X_1 X_2 \in$ a nejméně jeden ze symbolů X_1, X_2 je z , pak
$A \to X_1' X_2' \in$;
(5) pro každý neterminál ′ (zavedený v krocích (3) a (4)) zavedeme do pravidlo ′ → .

Nechť ′ obsahuje a všechny nové neterminály zavedené v krocích (3) a (4). Takto definovaná gramatika ′ je jistě v Chomského formě; důkaz, že

( ) = (′)

nalezne čtenář např. v [2].

DEF

Definice 2.5.19.

Bezkontextová gramatika = (, , ,) je v Greibachové normální formě, jestliže je bez ε-pravidel (viz definice 2.5.10) a každé pravidlo z různé od $\to \varepsilon$, je tvaru

$$A \to x$$

Kde ∈ , $x \in$ *, $A \in$.

Věta 2.5.20.

Buď bezkontextový jazyk. Pak existuje bezkontextová gramatika v Greibachové normální formě tak, že = ( ). Důkaz tohoto tvrzení nalezneme ku příkladu v [21].

2.5.2 Zásobníkové automaty

Zásobníkový automat tvoří teoretický model syntaktického analyzátoru, a proto mu budeme věnovat poměrně značnou pozornost. Nejprve si jej definujeme:

DEF

Definice 2.5.20.

Zásobníkový automat je sedmice

$$A = (Q, \ , Z, \delta, q_0, Z_0, F)$$

kde Q je neprázdná konečná množina stavů, je neprázdná vstupní abeceda, Z je neprázdná zásobníková abeceda, $q_0 \in Q$ je počáteční stav, $Z_0 \in Z$ je počáteční zásobníkový symbol, $F \subseteq Q$ je množina koncových stavů, δ je zobrazení $Q \times (\cup \{\varepsilon\}) \times Z$ do množiny všech konečných podmnožin $Q \times Z^*$; δ se nazývá přechodová funkce. Konfigurace zásobníkového automatu A je trojice

$$(q, \ , \ )$$

kde $q \in Q$, $\in$ *, $\in Z^*$. Řekneme, že konfigurace (q, , Y) přímo (též bezprostředně) přejde do konfigurace (p, ,), symbolicky

$$(q, \quad , Y\ ) \vdash (p, \quad , \quad)$$

pro nějaké $\in$ $\cup \{\varepsilon\}$, jestliže

$$(p, \quad) \in \delta\,(q, \quad , Y).$$

Konfigurace automatu A přejde do konfigurace ', symbolicky

$$\vdash^* \ ',$$

právě když existují konfigurace $_1, \ldots, \ _n$ (pro nějaké celé $n \geq 2$) tak, že

(1) $= \ _1$,
(2) $_i \vdash \ _{i+1}$, $1 \leq i \leq n$-1,
(3) $' = \ _n$,

nebo

$$= \ '.$$

Relace $\vdash^*$ je tzv. tranzitivní a reflexní uzávěr (viz [14]) relace $\vdash$. Jazyk akceptovaný automatem A, symbolicky (A), definujeme jako

$$(A) = \{ \ : (q_0, \quad , Z_0) \vdash^* (q, \varepsilon, \ ) \text{ pro nějaké } q \in F, \ \in Z^*\}.$$

Jazyk akceptovaný automatem A s prázdným zásobníkem, symbolicky $_\varepsilon(A)$, definujeme:

$$_\varepsilon(A), = \{ \ : (q_0, \quad , Z_0) \vdash^* (q, \varepsilon, \varepsilon) \text{ pro nějaké } q \in Q\}.$$

Podobně jako konečný automat má i tedy zásobníkový automat konečně-stavovou řídicí jednotku a na vstupní pásce má zapsané vstupní slovo, které čte pomocí čtecí hlavy. Navíc je však vybaven (potencionálně nekonečnou) tzv. zásobníkovou pamětí (krátce: zásobníkem), jejíž vrchol (dohodněme se, že se bude jednat o první symbol zleva) ovlivňuje každý přímý přechod (místo "přímý přechod" budeme často užívat termínu "takt") zásobníkového automatu.

Všimněme si, že konfigurace zásobníkového automatu nám vlastně popisuje jeho stávající "situaci", tj. stav, ve kterém se automat nachází, úsek vstupního slova, který má automat ještě projít a aktuální obsah zásobníku. Skutečnost, že automat přímo přejde (tj. provede jeden takt) z konfigurace (q, , Y) do konfigurace (p, ,) (nechť všechny symboly mají stejný význam jako v definici 2.5.20), tj.

$$(q, \quad , Y\ ) \vdash (p, \quad , \quad)$$

znamená, že

(i) automat přejde ze stavu q do stavu p,
(ii) symbol Y (tzn. vrchol zásobníku) bude nahrazen slovem ,
(iii) pokud symbol $\neq \varepsilon$, pak automat tento symbol přečte a posune čtecí hlavu na vstupní pásce o jedno pole doprava a pokud $= \varepsilon$, pak čtecí hlava daného automatu zůstává na místě.

Ilustrujme si nyní zásobníkový automat příkladem:

x+y

Příklad 2.5.21.
Uvažujeme zásobníkový automat

$$A = (\{q_1, q_2\}, \{0, 1\}, \{Z_0, Y\}, \delta, q_1, Z_0, \varnothing)$$

kde δ je definován takto:

$$\delta(q_1, 0, Z_0) = \{(q_1, YZ_0)\},$$
$$\delta(q_1, 0, Y) = \{(q_1, YY)\},$$
$$\delta(q_1, 1, Y) = \{(q_2, \varepsilon)\},$$
$$\delta(q_2, 1, Y) = \{(q_2, \varepsilon)\},$$
$$\delta(q_2, \varepsilon, Z_0) = \{(q_2, \varepsilon)\}.$$

Snadno se přesvědčíme, že

$$_{\varepsilon}(A) = \{0^n 1^n : n \geq 1\},$$

což je (viz [2]) známý bezkontextový jazyk, který není regulární.

Mimo jazyka generovaného zásobníkovým automatem jsem v definici 2.5.20 zavedli ještě jazyk generovaný zásobníkovým automatem s prázdným zásobníkem. Ukažme si nyní, že obě tyto definice jazyka jsou ekvivalentní.

Věta 2.5.22.
Jazyk $=$ $_{\varepsilon}(A_1)$ pro nějaký zásobníkový automat A_1, právě když $=$ (A_2) pro nějaký zásobníkový automat A_2.

Důkaz.
(1) Buď

$$A_2 = (Q, \ , Z, \delta, q_0, Z_0, F)$$

libovolný zásobníkový automat. Sestrojíme zásobníkový automat A_1 tak, že

$$_{\varepsilon}(A_1) = \ (A_2).$$

Nechť

$$A_1 = (Q \cup \{q_0, p\}, \quad , Z \cup \{\text{♞}\}, \delta', p, \text{♞}, \varnothing),$$

kde δ' je definováno takto:

(i) $\delta'(p, \varepsilon, \text{♞}) = \{(q_0, Z_0, \text{♞})\}$,

(ii) pro každé q, $\bar{q} \in Q$, $\in \cup \{\varepsilon\}$, $D \in Z$:
$(\bar{q}, \ ) \in \delta'(q, \ , D)$, právě když $(\bar{q}, \ ) \in (q, \ , D)$,

(iii) pro každí $q \in F$, $D \in Z \cup \{\text{♞}\}$:
$(q_\varepsilon, \varepsilon) \in \delta'(q, \varepsilon, D)$,

(iv) pro každé $D \in Z \cup \{\text{♞}\}$:
$\delta'(q, \varepsilon, D) = \{(q_\varepsilon, \varepsilon)\}$.

A_1 nejprve přejde pomocí pravidla (i) do počáteční konfigurace automatu A_2, jenž má pod vrcholem zásobníku Z_0 symbol ♞. Prostřednictvím pravidel z (ii) simuluje A_1 činnost A_2. Jestliže se A_2 dostane do koncového stavu, pak A_1 může pomocí pravidla z (iii) přejít do stavu q_ε a pak (pomocí pravidel z (iv)) vyprázdnit zásobník. Symbol ♞ je v A_1 uložen pod Z_0 pro případ, kdyby se při simulaci výpočtu A_2 automatem A_1 vyprázdnil na konci výpočtu zásobník, což by znamenalo, že je vstupní slovo přijato. Tomuto zabráníme právě symbolem ♞, jenž je ze zásobníku odstraněn až po ukončení simulace A_2 automatem A_1. Zřejmě tedy

$$(A_2) = \ _\varepsilon(A_1)$$

(2) Buď

$$A_1 = (Q \ , \quad , Z \ , \delta \ , q_0 \ , Z_0 \ , \varnothing)$$

zásobníkový automat. Sestrojme A_2 tak, že

$$_\varepsilon(A_1) = \ (A_2).$$

Nechť

$$A_2 = (Q \cup \{p, q_f\}, \quad , Z \cup \{\text{♞}\}, \delta', \text{♞}, \{q_f\}),$$

kde δ' definujeme takto:

(i) $\delta'(p, \varepsilon, \text{♞}) = \{(q_0, Z_0, \text{♞})\}$,

(ii) pro každé $q \in Q$, $\in \cup \{\varepsilon\}$, $D \in Z$: $\delta'(p, \ , Z) = \delta(q, \ , Z)$,

(iii) $\delta'(q, \varepsilon, \text{♞}) = (q_f, \varepsilon)$ pro každé $q \in Q$.

Automat A_2 nejprve pomocí pravidla (i) přejde do počáteční konfigurace automatu A_1, jenž má pod vrcholem zásobníku Z_0 ještě nový symbol ♞. Pravidla z (ii) pak simulují činnost automatu A_1. V momen

om.mn m34.826571( )2.74(
záá.293142(m)-12.4659(e)3.7 0 10 (0)Tj /R12 12 Tf 0.5855 0 10.22( )-150.235(z41 0 0 1 359 0 10

uložených na zásobníku. Budeme však nyní předpokládat, že vrchol zásobníku není nejlevější, nýbrž nejpravější symbol zásobníkové paměti.

DEF

Definice 2.5.23.

Rozšířený zásobníkový automat nazveme sedmici

$$A = (Q, \quad , Z, \delta, q_0, Z_0, F),$$

kde δ je zobrazení konečné podmnožiny množiny $Q \times (\quad \cup \{\varepsilon\}) \times Z^*$ do množiny konečných podmnožin množiny $Q \times Z^*$ a kde všechny ostatní symboly mají stejný význam jako v definici 2.5.20. Podobně jako ve zmíněné definici je definována i konfigurace automatu A. Konfigurace $(q, \quad , \quad)$ přímo přejde do konfigurace $(p, \quad , x \ )$, symbolicky

$$(q, \quad , \quad) \vdash (p, \quad , x \ ),$$

kde $q, p \in Q, \quad \in \quad \cup \{\varepsilon\}, \quad \in \quad ^*, \quad , x, \quad \in Z^*$, jestliže

$$(p, x) \in \delta(q, \quad , \quad).$$

Jazyk akceptovaný A, symbolicky (A), je definován analogicky jako v definici 2.5.20.

Každý přímý přechod z jedné konfigurace do druhé se jako obvykle nazývá takt daného rozšířeného zásobníkového automatu. Povšimněme si, že automat z definice 2.5.23 může obecně provést takt i tehdy, když má zásobník prázdný, což zásobníkový automat z definice 2.5.20 neumožňoval.

x+y

Příklad 2.5.24.

Buď

$$\begin{aligned}
&A = (\{p, q\}, \{ \ , \ \}, \{ \ , \ , \ , \overline{Z} \}, \delta, q, \overline{Z}, \{p\}),\\
&\delta(q, \ , \varepsilon) = \{(q, \ )\},\\
&\delta(q, \ , \varepsilon) = \{(q, \ )\},\\
&\delta(q, \varepsilon, \varepsilon) = \{(q, \ )\},\\
&\delta(q, \varepsilon, \ ) = \{(q, \ )\},\\
&\delta(q, \varepsilon, \ ) = \{(q, \ )\},\\
&\delta(q, \varepsilon, \overline{Z}) = \{(p, \varepsilon)\}.
\end{aligned}$$

Tento rozšířený zásobníkový automat akceptuje jazyk

$$(A) = \{ \quad : \quad \in \{ \ , \ \}^*\}.$$

Uvažme si například, jak bude vypadat akceptace slova :

$$\begin{aligned}
(q, \quad , \overline{Z}) &\vdash (q, \quad , \overline{Z})\\
&\vdash (q, \quad , \overline{Z})\\
&\vdash (q, \quad , \overline{Z})\\
&\vdash (q, \quad , \overline{Z})\\
&\vdash (q, \quad , \overline{Z})\\
&\vdash (q, \quad , \overline{Z})\\
&\vdash (q, \quad , \overline{Z})
\end{aligned}$$

$\vdash (q, \ , \ \overline{Z})$
$\vdash (q, \varepsilon, \ \overline{Z})$
$\vdash (q, \varepsilon, \overline{Z})$
$\vdash (q, \varepsilon, \varepsilon).$

Mezi zásobníkovými automaty a rozšířenými zásobníkovými automaty existuje velmi těsná souvislost, jak ukazuje následující tvrzení.

Věta 2.5.25.
Jazyk je akceptován nějakým zásobníkovým automatem, právě když existuje rozšířený zásobníkový automat A tak, že = (A).

Důkaz: Viz např. [2].

Tento odstavec uzavřeme definicí deterministického zásobníkového a deterministického rozšířeného zásobníkového automatu.

DEF

Definice 2.5.26.
Zásobníkový automat

$$A = (Q, \ , Z, \delta, q_0, Z_0, F),$$

je deterministický, jestliže pro každé $(q, \ , \ ) \in Q \times (\ \cup \{\varepsilon\}) \times Z$ platí:

(1) $\delta(q, \ , Z)$ obsahuje nejvýše jeden prvek:
(2) jestliže $\delta(q, \varepsilon, \ ) \neq \varnothing$, pak $\delta(q, \ , \ ) = \varnothing$ pro všechna $\in$.

Rozšířený zásobníkový automat $A = (Q, \ , Z, \delta, q_0, Z_0, F)$ je deterministický, platí-li:

(1) pro každé $q \in Q$, $\in \ \cup \{\varepsilon\}$, $\in Z^*$ obsahuje $\delta(q, \ , \ )$ nejvýše jeden prvek;
(2) jestliže $\delta(q, \ , \ ) \neq \varnothing$ a $\delta(q, \ , r) \neq \varnothing$, $\in \ \cup \{\varepsilon\}$, $\neq r$, pak není prefix r, ani r není prefix (tj. $r \neq$ nebo $\neq r$ pro nějaké $\in Z^*$).

Z praktického hlediska mají deterministické (rozšířené) zásobníkové automaty a na jejich bázi definované deterministické (rozšířené) zásobníkové převodníky stěžejní význam (srov. kapitolu 3.3). Žel, pro některé bezkontextové jazyky není možné sestrojit deterministický zásobníkový automat, který by je akceptoval, viz [14].

2.5.3. Vztah zásobníkových automatů k bezkontextovým gramatikám a základní principy syntaktické analýzy.

Nyní dokážeme základní výsledek, který říká, že třída jazyků akceptovaných zásobníkovými automaty je právě třída bezkontextových jazyků.

Lemma 2.5.27.
Buď = (, , ,) bezkontextová gramatika. Pak lze sestrojit zásobníkový automat A tak, že ( ) = $\varepsilon(A)$.

Důkaz. Buď

$$= (, , ,)$$

Libovolná bezkontextová gramatika. Sestojíme zásobníkový automat A tak, že bude modelovat všechny levé derivace v .

Buď

$$\mathrm{A} = (\{q\}, , \cup , \delta, q, , \varnothing),$$

kde přechodová funkce δ je definována takto:

(i) jestliže $A \to x \in$
pak $(q, x) \in \delta(q, \varepsilon, A)$;
(ii) $\delta(q, ,) = \{(q, \varepsilon)\}$ pro každé $\in$.

Tím je celá konstrukce dokončena. Formální dokončení důkazu, že vskutku

$$\varepsilon(A) = ( )$$

nalezneme ku příkladu v [2].

Zásobníkový automat A z důkazu lemmatu 2.5.27 tvoří teoretický model jednoho důležitého typu syntaktického analyzátoru. Přestože se tedy metodami syntaktické analýzy budeme podrobně zabývat až později (viz kap. 3.3), uveďme si již na tomto místě několik předběžných poznámek.

Co je úkolem syntaktické analýzy v procesu překladu? Rozhodnout, zda daný vstupní řetěz (reprezentující zdrojový program) patří do příslušného zdrojového jazyka a pokud ano, sestrojit k němu derivační strom. Zásobníkový automat A z důkazu lemmatu 2.5.27 tvoří teoretický model syntaktického analyzátoru, jenž provádí zmíněnou konstrukci derivačního stromu shora dolů, tj. konstrukci derivačního stromu začíná od kořene a postupuje (zleva doprava) dolů, směrem ke vstupnímu řetězu. Lze to vyjádřit i jinak: Automat A hledá levou derivaci dané vstupní věty. Ilustrujme si to příkladem.

x+y

Příklad 2.5.28.
Buď

$$= (\{E, , F\}, \{ , *, +, (,)\}, , E)$$

bezkontextová gramatika, kde obsahuje pravidla:

(1) $E \to E +$,
(2) $E \to$,

(3) $\rightarrow$ $* F$,
(4) $\rightarrow F$,
(5) $F \rightarrow (E)$,
(6) $F \rightarrow$.

Podle konstrukce uvedené v důkazu lemmatu 2.5.27 definujeme zásobníkový automat:

$$A = (\{q\}, \{\ , *, +, (,)\}, \{E,\ , F,\ , *, +, (,)\}, \delta, q, E, \varnothing),$$

kde

$$\delta(q, \varepsilon, E) = \{(q, E +\ ), (q,\ )\},$$
$$\delta(q, \varepsilon,\ ) = \{(q,\ * F), (q, F)\},$$
$$\delta(q, \varepsilon, F) = \{(q, (E)), (q,\ )\},$$
$$\delta(q, \text{▯}, \text{▯}) = \{(q, \varepsilon)\} \text{ pro každé } \text{▯} \in \{\ , *, +, (,)\}.$$

Uvažujme nyní větu

$$+\ *\ \in\ (\ ),$$

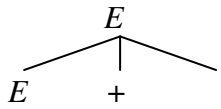
které odpovídá levá derivace (na konci řádku uvádíme číslo aplikovaného pravidla):

$E \Rightarrow E +$ (1)
$\Rightarrow\ +$ (2)
$\Rightarrow F +$ (4)
$\Rightarrow\ +$ (6)
$\Rightarrow\ +\ * F$ (3)
$\Rightarrow\ + F * F$ (4)
$\Rightarrow\ +\ * F$ (6)
$\Rightarrow\ +\ *$ (6)

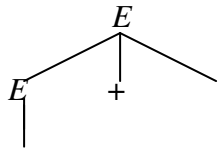
Automat A, jenž jsme výše zkonstruovali, provede akceptaci věty $+\ *$ takto (na konci řádku je původní číslo pravidla, kterému odpovídá příslušný takt):

$(q,\ +\ *\ , E) \vdash (q,\ +\ *\ , E +\ )$ (1)
$\vdash (q,\ +\ *\ ,\ +\ )$ (2)
$\vdash (q,\ +\ *\ , F +\ )$ (4)
$\vdash (q,\ +\ *\ ,\ +\ )$ (6)
$\vdash (q, +\ *\ , +\ )$
$\vdash (q,\ *\ ,\ )$
$\vdash (q,\ *\ ,\ * F)$ (3)
$\vdash (q,\ *\ , F * F)$ (4)
$\vdash (q,\ *\ ,\ * F)$ (6)
$\vdash (q, *\ , * F)$
$\vdash (q,\ , F)$
$\vdash (q,\ ,\ )$ (6)
$\vdash (q, \varepsilon, \varepsilon)$

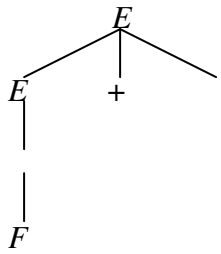
Výše uvedená levá derivace a posloupnost taktů automatu *A* odpovídá konstrukci derivačního stromu pro větu + * shora dolů (napravo budeme opět uvádět číslo aplikovaného pravidla):



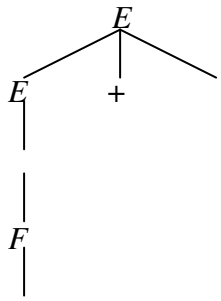
(1)



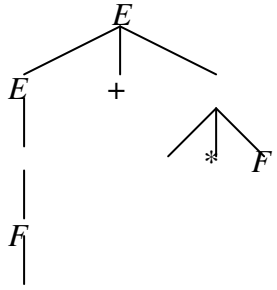
(2)



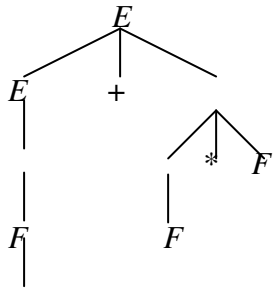
(4)



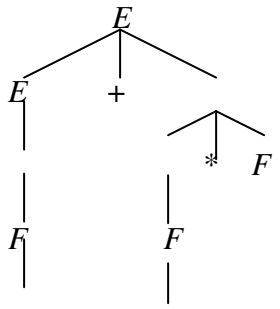
(6)



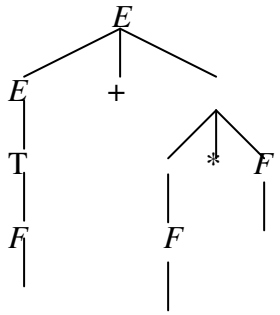
(3)



(4)



(6)



Z právě uvedeného příkladu by tedy mělo být zcela zřejmé, jak zásobníkový automat zkonstruovaný v důkazu lemmatu 2.5.27 může sloužit jako model syntaktického analyzátoru, který pracuje metodou shora dolů.

Nyní se zabývejme vztahem rozšířených zásobníkových automatů a bezkontextových gramatik.

Lemma 2.5.29.
Každý bezkontextový jazyk lze akceptovat vhodným rozšířeným zásobníkovým automatem.

Důkaz. Buď bezkontextový jazyk generovaný bezkontextovou gramatikou

$$= (\quad , \quad , \; , \;)$$

Definujme rozšířený zásobníkový automat

$$A = (\{q, r\}, \quad , \quad \cup \quad \cup \{\$\}, \delta, q, \$, \{r\}),$$

kde δ je definováno takto:

(1) $\delta(q, \quad , \varepsilon) = \{(q, \quad)\}$ pro každé $\in$,
(2) $\delta(q, \varepsilon, x) = \{(q, A): A \rightarrow x \in \; \}$,
(3) $\delta(q, \varepsilon, \$) = \{(r, \varepsilon)\}$.

Takto definovaný rozšířený zásobníkový automat akceptuje jazyk . Formální důkaz, že

$$(A) = \quad (\quad)$$

nalezneme např. v [2].

Rozšířený zásobníkový automat A z důkazu lemmatu 2.5.29 tvoří teoretický model dalšího významného typu syntaktického analyzátoru, jenž provádí konstrukci derivačního stromu (dané věty bezkontextového jazyka) zdola nahoru, tj. konstrukci začíná od vstupní věty, kterou se snaží (zleva doprava) redukovat na počáteční symbol. Akceptor A tedy vlastně hledá v obráceném pořadí pravou derivaci příslušné věty. Ilustrujeme příkladem.

x+y

Příklad 2.5.30.
Uvažujeme opět bezkontextovou gramatiku z příkladu 2.5.28 a větu

$$+ \quad * \quad \in \quad (\quad).$$

Sestrojíme nejprve pravou derivaci této věty (stejně jako v příkladu 2.5.28 uvádíme na konci řádků čísla aplikovaných pravidel):

$$\begin{aligned} E &\Rightarrow E + && (1)\\ &\Rightarrow E + \quad * F && (3)\\ &\Rightarrow E + \quad * && (6)\\ &\Rightarrow E + F * && (4)\\ &\Rightarrow E + \quad * && (6)\\ &\Rightarrow \quad + \quad * && (2)\\ &\Rightarrow F + \quad * && (4)\\ &\Rightarrow \quad + \quad * && (6) \end{aligned}$$

Nyní sestrojme podle důkazu lemmatu 2.5.29 rozšířený zásobníkový automat A, jenž bude akceptovat ( ).

$$A = (\{q, r\}, \{+, *, (,), \quad\}, \{E, \quad, F, +, *,), (, \quad, \$\}, \delta, q, \$, \{r\})$$

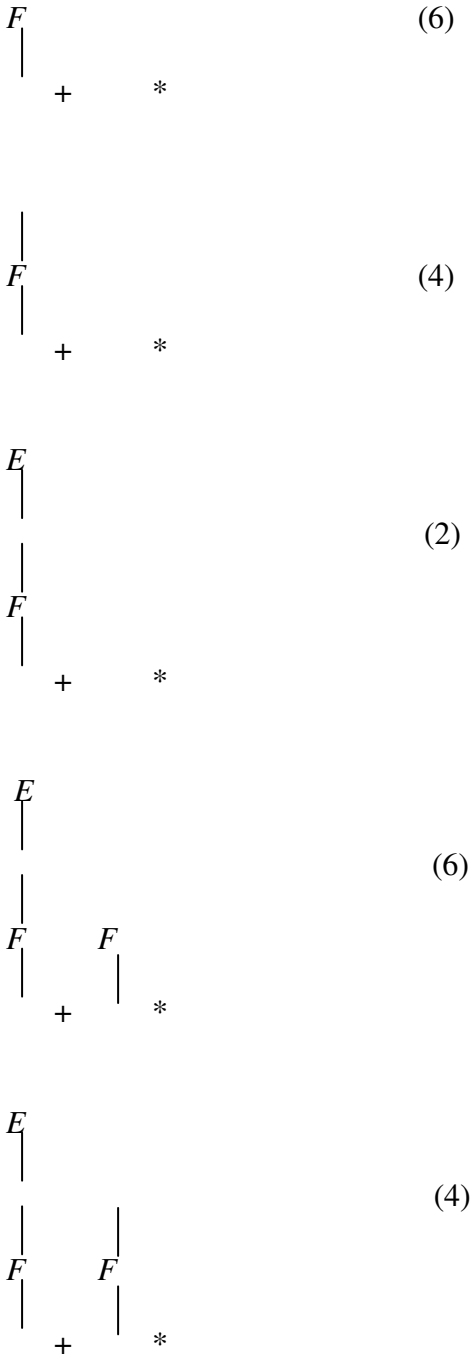
$\delta(q, \flat, \varepsilon) = \{(q, \flat)\}$ pro každé $\flat \in \{+, *, (,), \quad\}$,
$\delta(q, \varepsilon, E + \quad) = \{(q, E)\}$,
$\delta(q, \varepsilon, \quad) = \{(q, E)\}$,
$\delta(q, \varepsilon, \quad * F) = \{(q, \quad)\}$,
$\delta(q, \varepsilon, F) = \{(q, \quad)\}$,
$\delta(q, \varepsilon, (E)) = \{(q, F)\}$,
$\delta(q, \varepsilon, \quad) = \{(q, F)\}$,
$\delta(q, \varepsilon, \$E) = \{(r, \varepsilon)\}$.

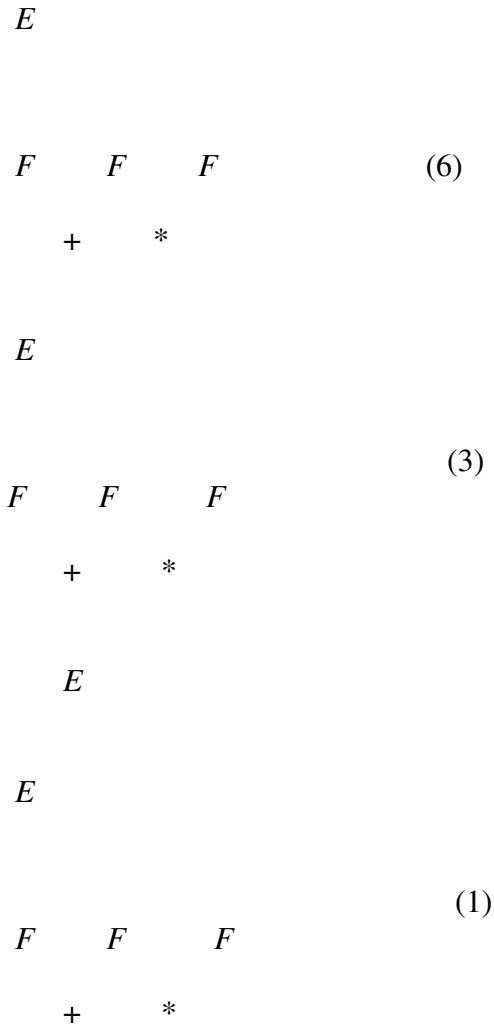
Uvidíme, že akceptace věty + * automatem A odpovídá pravé derivaci této věty sestrojené v obráceném pořadí (opět budeme napravo uvádět původní čísla pravidel, kterým odpovídá prováděný takt):

$$\begin{aligned} (q, \quad + \quad * \quad, \$) &\vdash (q, + \quad * \quad, \$ \,) && \\ &\vdash (q, + \quad * \quad, \$F) && (6)\\ &\vdash (q, + \quad * \quad, \$ \,) && (4)\\ &\vdash (q, + \quad * \quad, \$E) && (2)\\ &\vdash (q, \quad * \quad, \$E+) && \\ &\vdash (q, * \quad, \$E+ \,) && \end{aligned}$$

$\vdash$ $(q, *\ \ , \$E+F)$ (6)
$\vdash$ $(q, *\ \ , \$E+\ )$ (4)
$\vdash$ $(q, \ \ , \$E+\ *)$
$\vdash$ $(q, \varepsilon, \$E+\ *\ )$
$\vdash$ $(q, \varepsilon, \$E+\ *F)$ (6)
$\vdash$ $(q, \varepsilon, \$E+\ )$ (3)
$\vdash$ $(q, \varepsilon, \$E)$ (1)
$\vdash$ $(r, \varepsilon, \varepsilon)$

Postupně pomocí pravidel (6), (4), (2), (6), (4), (6), (3), (1) můžeme nyní sestrojit zdola nahoru derivační strom pro větu + * :





Jak je z právě uvedeného příkladu 2.5.30 patrné, rozšířený zásobníkový automat může skutečně sloužit jako teoretický model syntaktického analyzátoru, který pracuje zdola nahoru.

Všimněme si, že oba modely pro syntaktické analyzátory byly obecně nedeterministické. Z praktických důvodů přirozeně požadujeme, aby modelem syntaktického analyzátoru pracujícího shora dolů byl deterministický zásobníkový automat a modelem syntaktického analyzátoru pracujícího zdola nahoru byl deterministický rozšířený zásobníkový automat. Žel, ne vždy toho lze dosáhnout, neboť existují bezkontextové jazyky, jež nelze akceptovat žádným deterministickým zásobníkovým automatem resp. deterministickým rozšířeným zásobníkovým automatem. Nicméně existuje třída tzv. LL resp. LR gramatik, pro které lze vždy sestrojit deterministický syntaktický analyzátor pracující shoda dolů, resp. zdola nahoru. Těmito gramatikami se budeme zabývat až v odstavci 3.3.8 resp. 3.3.10.

Lemma 2.5.31

Bnit itad

Důkaz. Bezkontextovou gramatiku

$$= (\quad , \quad , \quad , \quad)$$

definujeme takto:

(1) $= \{[qBr]: q, r \in Q, B \in Z\} \cup \{\quad\}$;
(2) Pravidla z jsou definována:
   (a) jestliže $(r, X_1 \ldots X \quad) \in \delta\ (q, \quad , Z), X_i \in \quad \cup \quad , 1 \leq i \leq \quad , \quad \geq 1$, pak $[\ Z\$\ ] \rightarrow \quad [rX_1\$_1][\$_1X_2\$_2]\ldots[\$\ _{-1}X\ \$\ ] \in$ pro každou posloupnost $\$_1, \$_2, \ldots, \$\ \in Q$;
   (b) jestliže $(r, \varepsilon) \in \delta\ (q, \quad , Z)$, pak $[qZr] \rightarrow \quad \in$
   (c) pro každé $q \in Q$: $\rightarrow [q_0Z_0q] \in$

Formální důkaz, že

$$_{\varepsilon}(A) = \quad (\quad)$$

nalezneme opět v [2].

Výsledky tohoto odstavce můžeme nyní sumarizovat v následujícím tvrzení:

Věta 2.5.32.
Tvrzení

(1) $=$ ( ) pro nějakou bezkontextovou gramatiku ,
(2) $= (A_1)$ pro nějaký zásobníkový automat A_1,
(3) $= {}_{\varepsilon}(A_2)$ pro nějaký zásobníkový automat A_2,
(4) $= (A_3)$ pro nějaký rozšířený zásobníkový automat A_3,

jsou ekvivalentní.

Důkaz. Plyne z vět 2.5.25 a 2.5.22 a lemmat 2.5.27, 2.5.29, 2.5.31.

2.5.4. Cvičení

1. Převeďte bezkontextovou gramatiku

$$= (\{\quad, A, B\}, \{\quad, \quad\}, \quad, \quad),$$
$$= \{\quad \rightarrow A,$$
$$\quad \rightarrow B,$$
$$A \rightarrow \quad B,$$
$$A \rightarrow \quad ,$$
$$A \rightarrow$$
$$B \rightarrow AB$$
$$B \rightarrow B \quad ,$$
$$B \rightarrow A$$
$$B \rightarrow \ \}$$

na ekvivalentní redukovanou bezkontextovou gramatiku.

2. Buď

$$= (\{\ , A, B\}, \{\ ,\ \}, \ ,\ ),$$
$$= \{\ \ \rightarrow\ \ B,$$
$$\ \ \rightarrow\ A,$$
$$A \rightarrow\ \ ,$$
$$A \rightarrow\ AA,$$
$$A \rightarrow$$
$$B \rightarrow\ \ $$
$$B \rightarrow\ \ BB,$$
$$B \rightarrow\ \ \}$$

bezkontextová gramatika. Převeďte ji do Chomského normální formy.

3. Dokažte, že každý bezkontextový jazyk ($\varepsilon \notin$) lze generovat bezkontextovou gramatikou, jejíž libovolné pravidlo je jednoho z následujících tvarů :

$$A \rightarrow\ BC,$$
$$A \rightarrow\ B,$$
$$A \rightarrow\ ,$$

kde je terminál a A, B, C jsou neterminály.

4. Sestrojte zásobníkový automat akceptující doplněk následujících jazyků:

(a) $\{\ ^{n}\ ^{n}c^{n}: n \geq 1\}$,
(b) $\{\ \ :\ \ \in \{\ ,\ \}^{*}\}$,
(c) $\{\ \ ^{n}\ \ ^{n}: n, \geq 1\}$.

5. Nalezněte deterministický zásobníkový automat akceptující jazyk:

(a) $\{O^{i}\ ^{i}: j \leq i\}$,
(b) $\{\ \ c\ ^{\mathrm{R}}:\ \ \in \{\ ,\ \}^{+}\}$.

6. Buď

$$A = (\{q_0, q_1, q_2\}, \{\ ,\ \}, \{Z_0, A\}, \delta\ , q_0, Z_0, \{q_2\}),$$
$$\delta\ (q_0,\ \ , Z_0) = \{(q_1, AZ_0)\},$$
$$\delta\ (q_0,\ \ , A) = \{(q_1, AA)\},$$
$$\delta\ (q_1,\ \ , A) = \{(q_0, AA)\},$$
$$\delta\ (q_1, \varepsilon, A) = \{(q_2, A)\},$$
$$\delta\ (q_2, \ , A) = \{(q_2, \varepsilon)\}$$

zásobníkový automat. Nalezněte vlastní bezkontextovou gramatiku takovou, že ( ) = (A).

3. Kompilátory

Ve zbývající části tohoto textu si vyložíme základní principy konstrukce jednotlivých částí kompilátoru, tj. programu, jenž překládá zdrojový program na funkčně ekvivalentní cílový program. Tento překlad probíhá, jak jsme si již řekli v úvodu, ve třech hlavních fázích: lexikální analýza, syntaktická analýza a generování kódu. Každá z těchto fází je vlastně sama rovněž překladem: lexikální analyzátor překládá zdrojový program na řetěz lexikálních symbolů, který syntaktická analýza překládá na řetěz reprezentující derivační strom a tento řetěz je pak generátorem kódu přeložen na cílový program.

3.1. Formální překlad

Začneme definicí (formálního) překladu:

DEF

Definice 3.1.1.

Buďte V, V_O vstupní abecedy a mějme jazyky $L \subseteq V^*$, $L_O \subseteq V_O^*$.
Překladem z jazyka L do jazyka L_O nazveme relaci

$$\tau \subseteq L \times L_O.$$

V resp. V_O se nazývá vstupní resp. výstupní abeceda. Jestliže $(x, y) \in \tau$, pak slovo y nazýváme překlad věty x (nebo výstup pro větu x z překladu τ).

Všimněme si, že je překlad definován jako relace, nikoliv jako zobrazení. V obecném případě tedy může existovat pro danou vstupní větu několik výstupů. V praxi však pochopitelně požadujeme, aby pro každou větu (reprezentující zdrojový program) existoval nejvýše jeden překlad, tj. aby relace τ byla zobrazením.

Problém specifikace překladu je podobný jako problém specifikace (formálního) jazyka. Překlad je obvykle tvořen nekonečnou množinou dvojic slov a my potřebujeme nalézt konečné prostředky, které by nám tyto nekonečné množiny umožnily specifikovat. K nejznámějším a pro praxi nejvýznamnějším prostředkům patří syntaxí řízená překladová schémata.

DEF

Definice 3.1.2.

Syntaxí řízené překladové schéma je pětice

$$S = (N, V, V_O, P, S),$$

kde

N je abeceda neterminálů,
V je vstupní abeceda,
V_O je výstupní abeceda,
P je konečná množina pravidel tvaru $A \rightarrow x, y$, kde $A \in N$, $x \in (N \cup V)^*$, $y \in (N \cup V_O)^*$ a

přitom neterminály z jsou permutací neterminálů z x, je počáteční symbol,
∈ .

Mějme pravidlo $A \to x$, ∈ . Každému neterminálu v x je přidružen tentýž neterminál v . Jestliže se v x vyskytuje každý neterminál právě jednou, pak se vyskytuje právě jednou i v a přidružení je jasné. Vyskytuje-li se některý neterminál v x i vícekrát, pak užijeme horních indexů k určení vzájemného přidružení neterminálů, např.

$$A \to \overbrace{B^1 + \overbrace{B^2}, B^2 B^1}+.$$

Překladovou formu schématu definujeme následovně:

(1) (,) je překladová forma, ve které jsou si oba výskyty neterminálů přidruženy;
(2) jestliže (A , A) je překladová forma, oba uvedené výskyty A jsou přidružené neterminály, ∈ (∪)*, , ∈ (∪ $_O$)*, $A \to x$, ∈ , pak (x ,) je překladová forma, kde přidružení výskytů symbolů v x a je dáno jejich přidružením v pravidle $A \to x$, . V tomto případě také říkáme, že překladová forma (A , A) přímo derivuje překladovou formu (x ,), což symbolický zapisujeme

$$(A, A) \Rightarrow (x,).$$

Relace ⇒ se nazývá překladová derivace.

Řekneme, že překladová forma (,) derivuje překladovou formu (′, ′), (,) ∈ (∪)* × (∪ $_O$)*, symbolicky

$$(,) \Rightarrow^* (′, ′)$$

jestliže buď

$$(,) = (′, ′)$$

a nebo existují překladové formy:

$$(x_0, {}_0), (x_1, {}_1), \ldots, (x_n, {}_n), \text{ kde } n \geq 1$$

tak, že

(1) $(x_0, {}_0) = (,)$,
(2) $(x_n, {}_n) = (′, ′)$,
(3) $(x_{i-1}, {}_{i-1}) \Rightarrow (x_i, {}_i)$, $1 \leq i \leq n$.

Překlad definovaný si označíme) a definujeme jej:

$$) = \{(x,): (,) \Rightarrow^* (x,), x \in {}^*, \in {}_O^*\}.$$

x+y

Příklad 3.1.3.
Uvažujme syntaxí řízené překladové schéma

$$\quad) = (\{\quad\}, \{0, 1\}, \{0, 1\}, \quad, \quad),$$

kde obsahuje pravidla:

$$\quad \rightarrow 0\quad, \quad 0$$
$$\quad \rightarrow 1\quad, \quad 1$$
$$\quad \rightarrow \varepsilon, \varepsilon.$$

Velmi snadno se přesvědčíme, že definuje překlad

$$\quad) = \{(x, x^{R}): x \in \{0, 1\}^{*}\}.$$

Podívejme se ku příkladu na derivaci $(001, 100) \in$):

$$\begin{aligned}(\quad, \quad) &\Rightarrow (0\quad, \quad 0)\\ &\Rightarrow (00\quad, \quad 00)\\ &\Rightarrow (001\quad, \quad 100)\\ &\Rightarrow (001, 100). \square\end{aligned}$$

DEF

Definice 3.1.4.
Buď $= (\quad, \quad, \quad_{O}, \quad, \quad)$ syntaxí řízené překladové schéma. Pak) se nazývá syntaxí řízený překlad**.** Gramatika

$$= (\quad, \quad_{,} \quad, \quad),$$

kde

$$\quad = \{A \rightarrow x: A \rightarrow x, \quad \in \quad\}$$

se nazývá vstupní gramatika schématu a gramatika

$$_{O} = (\quad, \quad_{O,} \quad_{O}, \quad),$$

kde

$$\quad_{O} = \{A \rightarrow \quad : A \rightarrow x, \quad \in \quad\}$$

se nazývá výstupní gramatika schématu .

Syntaxí řízený překlad lze také chápat jako metodu transformace derivačního stromu ve vstupní gramatice na derivační strom ve výstupní gramatice $_{O}$. Mějme větu $x \in$ ( ). Překlad této věty lze určit tak, že sestrojíme derivační strom pro větu x v a transformujeme jej (pomocí

níže uvedeného algoritmu 3.1.5) na derivační strom ve výstupní gramatice $_O$. Tím získáváme nový derivační strom věty , kde je překlad věty x. Popišme si tuto transformaci algoritmem:

Algoritmus 3.1.5.

Transformace derivačního stromu.

Vstup: Syntaxí řízené překladové schéma = (, , $_O$, ,) se vstupní gramatikou
= (, , ,) výstupní gramatikou $_O$ = (, $_O$, $_O$,), a derivační strom D_x věty
$x \in$ * v .

Výstup: Derivační strom D věty $\in$ * ve výstupní gramatice $_O$ tak, že (x,) ∈).

Metoda:

(1) Aplikuj (2) na kořen D_x.

(2) Nechť tento krok je aplikován na uzel jenž není koncový a nechť , , … , jsou přímí následovníci uzlu jejichž ohodnocení (zleva doprava) tvoří řetěz x a buď $A \to x$, ∈ .
   (a) Odstraníme z D_x všechny koncové uzly;
   (b) Se zbývajícími uzly provedeme permutaci včetně příslušných podstromů tak, aby jejich pořadí odpovídalo pořadí neterminálů v ;
   (c) připojíme nové koncové uzly tak, aby ohodnocení všech přímých následovníků uzlu tvořilo řetěz ;
   (d) Krok (2) aplikujeme na všechny přímé následovníky uzlu které nejsou koncové uzly.

(3) Výsledný strom je D . □

Ilustrujme si tento algoritmus příkladem.

x+y

Příklad 3.1.6.

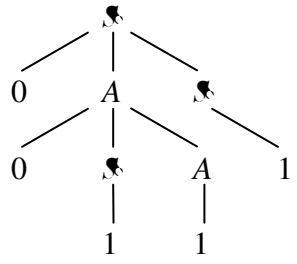
Mějme syntaxí řízené překladové schéma

= ({ , A}, {0, 1}, { , }, ,),

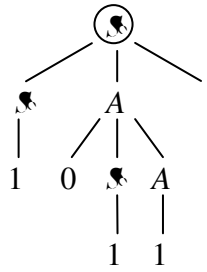
kde obsahuje pravidla

$\to 0A$, A
$A \to 0A$, A
$\to 1$,
$A \to 1$,

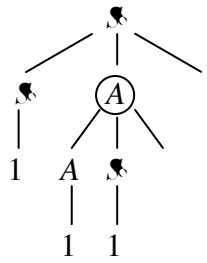
a uvažujme derivační strom



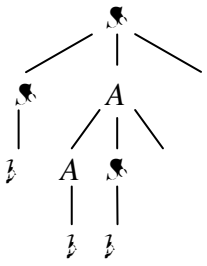
věty 00111 v .
Aplikujme krok (2) na kořen . Získáme derivační strom



Nyní aplikujme krok (2) na přímého následovníka kořene , tj. na uzel *A*. Získáváme strom



Kdybychom takto postupovali dál, získali bychom nakonec výsledný, transformovaný, derivační strom:



Tedy, (00111,) ∈). □

3.2. Lexikální analýza

3.2.1. Úvod

Zdrojový program si můžeme jednoduše představit jako řetěz znaků, jenž je zaznamenán na vhodném paměťovém mediu (např. na magnetickém disku, magnetické pásce, děrné pásce, disketě apod.). Odtud je pak čten první částí kompilátoru, tzn. lexikálním analyzátorem, který jej překládá na řetěz lexikálních symbolů (někdy se užívá místo lexikální symbol termínu atom, textový element, terminální symbol nebo jednoduše symbol), což jsou jednotlivé symboly zdrojového programu, jako např. identifikátory, rezervovaná slova, celá čísla atd.

Uvědomme si, že při rekonstrukci řetězu lexikálních symbolů se lexikální analyzátor musí vypořádat s celou řadou problémů:

Tak např. různá vstupní zařízení používají různé kódy, což je ovšem z pohledu vlastního překladu programu do cílového jazyka zcela irelevantní. Dále, stejné číslo lze v daném programovacím jazyce vyjádřit různým způsobem, přičemž samozřejmě pro přeložený program má význam pouze jeho hodnota. Důležitým úkolem lexikálního analyzátoru není proto pouze nalezení a rozpoznání jednotlivých lexikálních symbolů, ale i jejich vhodné (zpravidla celočíselné) zakódování, neboť syntaktický analyzátor pak pracuje se symboly reprezentovanými pomocí celých čísel a nikoli řetězy znaků proměnné délky, jak jsou uvedeny ve zdrojovém programu.

Poznamenejme, že lexikální analyzátor má zpravidla za úkol i další, méně podstatné úkoly, jako např. vynechání komentářů, popř. některých dalších textů zdrojového programu, které slouží pouze pro zvýšení čitelnosti či grafické úpravy programu a pro překlad vlastního programu jsou naprosto bezvýznamné.

Lexikální analyzátor může být skutečně naprogramován tak, že provede kompletní lexikální analýzu zdrojového programu a na výstupu vytváří řetěz lexikálních symbolů, jež je pak čten syntaktickým analyzátorem. V praxi se však takto realizuje lexikální analyzátor jen zřídka. Zpravidla je lexikální analyzátor naprogramován jako podprogram syntaktického analyzátoru, jenž jej vyvolá vždy, když potřebuje další lexikální symbol.

Většinu činností lexikální analýzy lze modelovat pomocí tzv. konečného převodníku. Proto mu budeme v této kapitole věnovat značnou pozornost.

3.2.2. Regulární syntaxí řízený překlad schémata

Nejprve si zaveďme jistý speciální případ syntaxí řízených překladových schémat, které mají, jak později poznáme, ke konečným převodníkům velmi úzký vztah.

DEF

Definice 3.2.1.

Buď

$$= (\quad _1 \quad _0 , ,)$$

syntaxí řízené překladové schéma a nechť každé pravidlo z má jeden z těchto dvou tvarů:

(1) $A \rightarrow \ B, xB,$
(2) $A \rightarrow \ , x,$

kde $\in \Sigma \cup \{\varepsilon\}$, $x \in \Delta^*$. Pak je regulární syntaxí řízené překladové schéma. Překlad), který definuje, nazýváme regulární překlad.

Následující příklad ukazuje, jak lze prostřednictvím regulárního syntaxí řízeného překladového schématu definovat překlad, jenž vytváří lichou paritu.

x+y

Příklad 3.2.2.
Mějme regulární syntaxí řízené překladové schéma

$$= (\{, A, B\}, \{0, 1\}, \{0, 1\}, ,),$$

kde obsahuje pravidla:

$\to 0A, 0A$
$\to 1B, 1B$
$A \to 0A, 0A$
$A \to 1B, 1B$
$A \to \varepsilon, 1$
$B \to 0B, 0B$
$B \to 1A, 1A$
$B \to \varepsilon, 0.$

Snadno se přesvědčíme, že skutečně
$) = \{(x, x_1): x \in \{0,1\}^*$, x obsahuje sudý počet výskytů symbolů $1\}$
$\cup \{(x, x_0): x \in \{0,1\}^*$, x obsahuje lichý počet výskytů symbolů $1\}$.

3.2.3. Konečné převodníky

Konečný převodník získáme poměrně jednoduše z konečného automatu a to tím, že ho rozšíříme o možnost výstupu řetězu symbolů (nad výstupní abecedou).

DEF

Definice 3.2.3
Konečný převodník budeme nazývat šestici

$$= (Q, \Sigma, \Delta, \delta, q_0, F)$$

kde
Q je konečná množina stavů,
Σ je vstupní abeceda,
Δ je výstupní abeceda,
δ je zobrazení $Q \times (\Sigma \cup \{\varepsilon\})$ do množiny konečných podmnožin $Q \times \Delta^*$,
q_0 je počáteční stav, $q_0 \in Q$,

F je konečná množina koncových stavů.

Konfigurací je trojice

$$(q, x, \ ) \in \ \ ^{*} \ _{O}^{*},$$

kde

q je stávající stav konečného převodníku ,
x je sufix vstupního řetězu, jenž doposud nebyl přečten,
je prefix výstupního řetězu, jenž doposud byl vytvořen.

Buď (q, x,), (r, x,) dvě konfigurace konečného převodníku , $q, r \in Q$, $\cup\{\varepsilon\}$, $x \in \ ^{*}$, , $\in \ _{O}^{*}$. Řekneme, že konfigurace (q, x,) přímo přejde do konfigurace (r, x,), symbolicky (q, x,) $\vdash$ (r, x,), tehdy a jen tehdy, jestliže $($$q$,) obsahuje ($r$,). Buďte C, C' konfigurace . Řekneme, že C přejde do C', symbolicky $C \vdash^{*} C'$, jestliže buď $C = C'$ a nebo existují konfigurace $C_0, ..., C$ (pro nějaké ≥ 1), tak že $C_{i-1} \vdash C_i$, $1 \leq i \leq$, $C_0 = C$, C $= C'$.

Konfigurace (q_0, x, ε), $x \in \ ^{*}$, se nazývá počáteční, konfigurace (q, ε,), kde $q \in F$, $\in \ _{O}^{*}$, se nazývá koncová.

$$(q_0, x, \varepsilon) \vdash^{*} (q, \varepsilon, \ ), \ q \in F .$$

Překlad) definovaný je množina dvojic

$$) = \{(x, \ ) : (q_0, x, \varepsilon) \vdash^{*} (q, \varepsilon, \ ), q \in F, x \in \ ^{*}, \in \ _{O}^{*}\}.$$

Všimněme si, že konečný převodník je obecně definován nedeterministicky. V porovnání s konečným automatem, dává konečný převodník možnost přechodu z jednoho stavu do druhého s prázdným vstupem (takovýto přechod nazýváme ε-přechod).

x+y

Příklad 3.2.4.
Buď

$$) = (\{ \ , \ , c, \ \}, \{0,1\}, \{ \ , \ , c, \ \}, \ , \ , \{ \ , c\})$$

konečný převodník se zobrazením :

$$(\ , \varepsilon) = \{(\ , \ )\},$$
$$(\ , 0) = \{(\ , \ )\},$$
$$(\ , 1) = \{(\ , \ )\},$$
$$(\ , 0) = \{(\ , \ )\},$$

$$
\begin{aligned}
\delta(\flat, 1) &= \{(c, c)\},\\
\delta(c, 0) &= \{(\ ,\ )\},\\
\delta(c, 1) &= \{(\ ,\ )\},\\
\delta(\ , 0) &= \{(\ ,\ )\},\\
\delta(\ , 1) &= \{(\ ,\ )\}.
\end{aligned}
$$

Tento konečný převodník definuje překlad, jenž reprezentuje posloupnost stavů, přes které přechází. Podívejme se na činnost konečného převodníku pro vstup 1001010:

$$
\begin{aligned}
(\ , 1001010, \varepsilon) &\vdash (\ , 1001010,\ )\\
&\vdash (\flat, 001010,\ \flat)\\
&\vdash (\flat, 01010,\ \flat\flat)\\
&\vdash (\flat, 1010,\ \flat\flat\flat)\\
&\vdash (c, 010,\ \flat\flat\flat c)\\
&\vdash (\ , 10,\ \flat\flat\flat c\ )\\
&\vdash (\ , 0,\ \flat\flat\flat c\ \ )\\
&\vdash (\ , \varepsilon,\ \flat\flat\flat c\ \ \ ).
\end{aligned}
$$

Výstupem pro vstupní řetěz 1001010 je řetěz $\flat\flat\flat c$.

DEF

Definice 3.2.5.

Konečný převodník $= (Q, V, V_O, \delta, q_0, F)$ budeme nazývat deterministický, jestliže pro všechny $q \in Q$ platí jedna z následujících dvou podmínek:

(1) $\delta(q,\ )$ obsahuje nejvýše jeden prvek pro každé $\in V$ a $\delta(q, \varepsilon) = \varnothing$,

a nebo

(2) pro každé $\in V$, $\delta(q,\ ) = \varnothing$ a $\delta(q, \varepsilon)$ obsahuje jeden prvek.

Poznamenejme, že deterministický konečný převodník může pro jediný vstup definovat několik výstupů, jak uvidíme v následujícím jednoduchém příkladě.

x+y

Příklad 3.2.6.

Buď

$$
\begin{aligned}
&= (\{q_0, q_1\}, \{\ \}, \{\flat\}, \delta, q_0, \{q_1\}),\\
&\delta(q_0,\ ) = \{(q_1, \flat)\},\\
&\delta(q_1,\ ) = \{(q_1, \flat)\}
\end{aligned}
$$

deterministický konečný převodník.
Pak zřejmě

$$(q_0, \ , \varepsilon) \vdash (q_1, \varepsilon, \ )$$
$$\vdash (q_1, \varepsilon, \ \ )$$
$$\vdash (q_1, \varepsilon, \ \ \ )$$
$$.$$
$$.$$
$$.$$

a tedy

$$\ ) = \{(\ , \ ^i) : i \geq 1\}.$$

3.2.4. Vztah regulárních syntaxí řízených překladových schémat ke konečným převodníkům

V tomto odstavci ukážeme, že mezi prostředky zavedenými v odstavcích 3.2.1 a 3.2.2 existuje těsná souvislost.

Věta 3.2.7.

Buď regulární syntaxí řízené překladové schéma. Pak existuje konečný převodník takový, že) =).

Důkaz. Mějme regulární syntaxí řízené překladové schéma

$$= (\ \ \ \ _O', \ , \ ).$$

Sestrojme konečný převodník

$$= (Q \ \ \ \ _O, q, \ , F)$$

takto:

(1) $Q = \ \cup \{X\}, X \notin \ ;$

(2) $\ $ je definováno:
- jestliže $A \rightarrow \ , \ \in$, pak $(X, \ ) \in (A, \ )$,
- jestliže $A \rightarrow \ B, \ B \in$, pak $(B, \ ) \in (A, \ )$,
pro každé $\in \ \cup \{\varepsilon\}, \in \ _O^*, A, B \in \ ;$

(3) $F = \{, X\}$, jestliže $\rightarrow \varepsilon, \ \in$,
$F = \{X\}$, jestliže $\rightarrow \varepsilon, \ \notin$.

Indukcí podle délky překladové derivace v a počtu přechodů v lze velmi snadno dokázat rovnost

$$) = \ ),$$

což je ponecháno na čtenáři (viz též [2]).

Vraťme se nyní k příkladu 3.2.2 a pokusme se dle konstrukce uvedené v důkazu věty 3.2.7 sestrojit konečný převodník , jenž by také vytvářel lichou paritu.

x+y

Příklad 3.2.8.

Uvažujme regulární syntaxí řízené překladové schéma z příkladu 3.2.2, tj.

$$= (\{\clubsuit, A, B\}, \{0,1\}, \{0,1\}, \clubsuit, \clubsuit),$$

kde $\clubsuit$ obsahuje pravidla

$$\begin{aligned}
\clubsuit &\to 0A,\ 0A \\
\clubsuit &\to 1B,\ 1B \\
A &\to 0A,\ 0A \\
A &\to 1B,\ 1B \\
A &\to \varepsilon,\ 1 \\
B &\to 0B,\ 0B \\
B &\to 1A,\ 1A \\
B &\to \varepsilon,\ 0.
\end{aligned}$$

Užitím konstrukce z důkazu věty 3.2.7 dospějeme ke konečnému převodníku

$$= (\{\clubsuit, A, B, X\}, \{0,1\}, \{0,1\}, \delta, q_0, \{X\}),$$

$$\begin{aligned}
\delta(\clubsuit, 0) &= \{(A, 0)\}, \\
\delta(A, 0) &= \{(A, 0)\}, \\
\delta(A, 1) &= \{(B, 1)\}, \\
\delta(A, \varepsilon) &= \{(X, 1)\}, \\
\delta(\clubsuit, 1) &= \{(B, 1)\}, \\
\delta(B, 0) &= \{(B, 0)\}, \\
\delta(B, 1) &= \{(A, 1)\}, \\
\delta(B, \varepsilon) &= \{(X, 0)\}.
\end{aligned}$$

Činnost konečného převodníku si ukažme pro vstupní řetězy 1011 a 1001:

(1.) $(\ , 1011, \varepsilon) \vdash (B, 011, 1) \vdash (B, 11, 10) \vdash (A, 1, 101) \vdash (B, \varepsilon, 1011) \vdash (X, \varepsilon, 10110)$

(2.) $(\ , 1001, \varepsilon) \vdash (B, 011, 1) \vdash (B, 01, 10) \vdash (B, 1, 100) \vdash (A, \varepsilon, 1001) \vdash (X, \varepsilon, 10011).$

Snadno se tedy přesvědčíme, že

$$\clubsuit) = \clubsuit).$$

3.2.5. Lexikální analyzátor

Hlavním úkolem lexikálního analyzátoru je nalézt, rozpoznat a zakódovat jednotlivé lexikální symboly zdrojového programu. V tomto odstavci poznáme, že jako vhodný model takovéhoto analyzátoru nám může sloužit obecný převodník.

Jak jsme již řekli v odstavci 2.4.2, lexikální symboly programovacích jazyků jsou nejčastěji popsány vhodným regulárním výrazem. Popišme si tedy pomocí nich několik symbolů a pokusme se k ním sestrojit konečný převodník, jež je bude rozpoznávat a kódovat.

Nejprve identifikátor (srov. Příklad 2.4.8):

$$\text{<IDENTIFIKATOR>} = P(C + P)^*,$$

kde C označuje číslici a P písmeno.

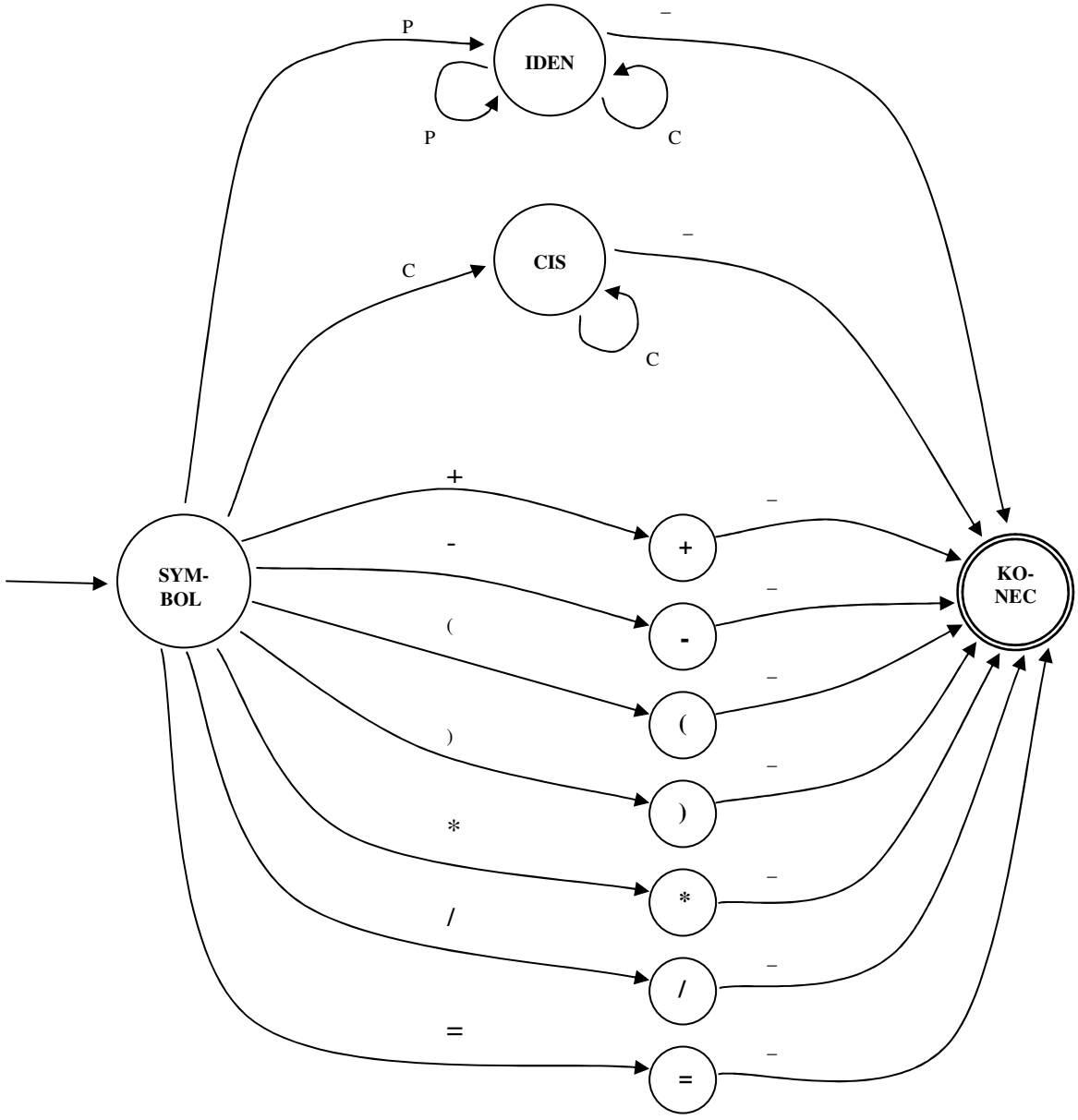
Nyní si definujeme lexikální symbol celé číslo:

$$\text{<CELE-CISLO>} = C\ (C)^*.$$

Zbývající lexikální symboly, které budeme rozpoznávat, jsou tyto oddělovače:

+, -, (,), *, /, =.

Pro jednoduchost předpokládejme, že každý lexikální symbol je ukončen speciálním znakem _ (mezera) a sestrojme si konečný automat (reprezentovaný stavovým diagramem), jenž uvedené lexikální symboly akceptuje:



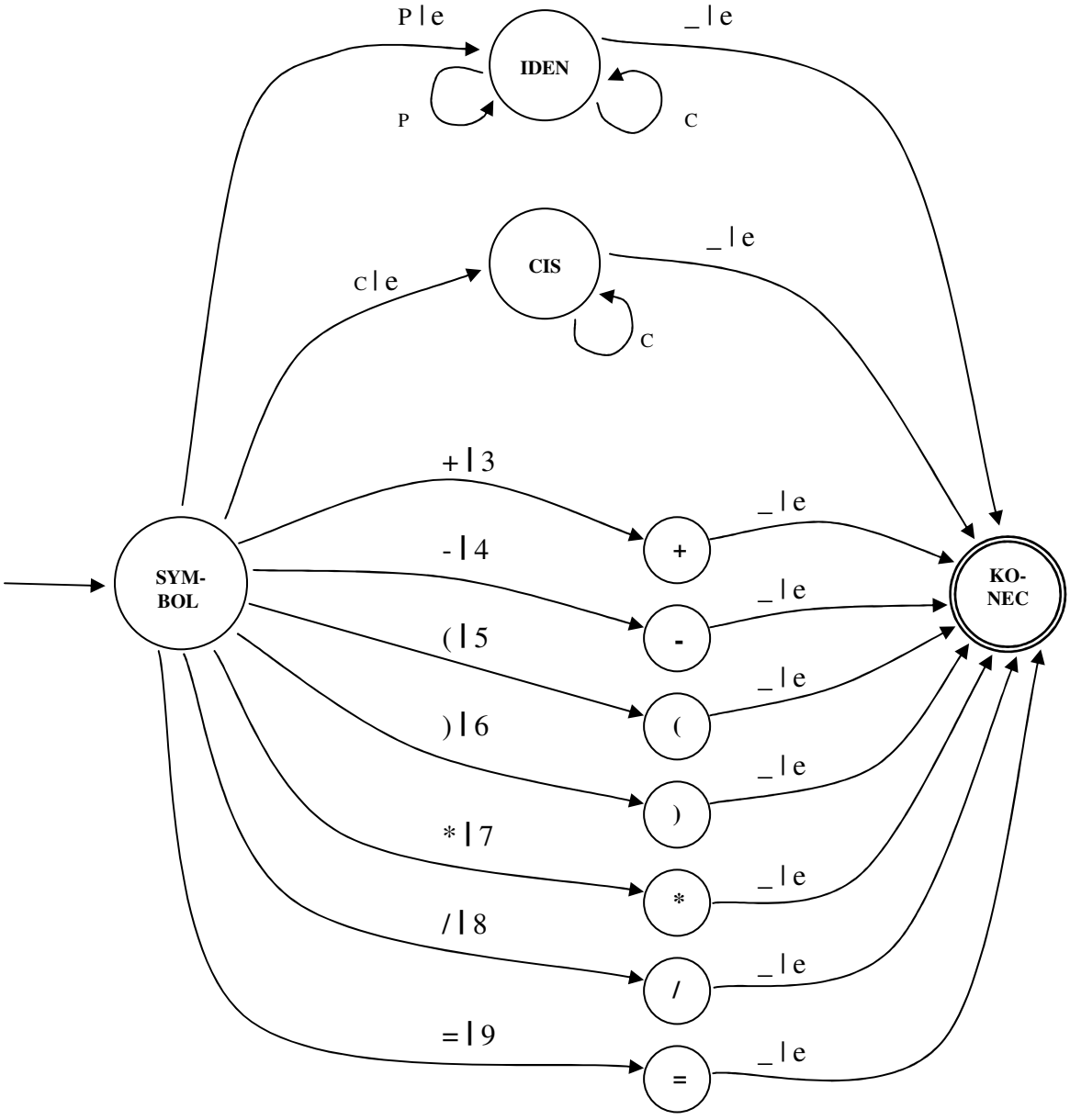
Lexikální symboly akceptované výše uvedeným konečným automatem chceme ovšem zakódovat pomocí celých čísel. Dohodněme se například, že jednotlivé symboly budeme kódovat takto:

lexikální symbol	kód
identifikátor	1
celé číslo	2
+	3
-	4
(	5
)	6
*	7
/	8
=	9

Je zřejmé, že pro zakódování lexikálních symbolů již nevystačíme s lexikálním automatem, který nemá možnost výstupu. Proto přejdeme ke konečnému převodníku, jenž okamžitě při rozpoznání příslušného lexikálního symbolu provede výstup jeho kódu.

Konečný převodník bude opět reprezentován stavovým diagramem, zcela analogicky jako konečný automat. Pouze hrany nyní budou ohodnoceny kromě vstupních symbolů i příslušnými výstupními řetězy (v našem případě bude ovšem každý výstupní řetěz tvořen jediným symbolem a to kódem lexikálního symbolu).

Hledaný konečný převodník kódující podle převodní tabulky jednotlivé lexikální symboly, je tento:



Jistě jsme si ale uvědomili, že jsme naše úvahy o lexikálním analyzátoru poněkud zjednodušili. Situace se například zkomplikuje, jestliže jednotlivé lexikální symboly nebudou odděleny speciálním znakem (v našem případě _), jak tomu také u řady programovacích jazyků je. Dále je

třeba rozpoznávat a kódovat víceznakové oddělovače (zvažme jen problém rozpoznávání lexikálním symbolů “*” a “**“). Ve zdrojovém textu programu by bylo také potřeba vynechávat nepotřebné symboly, různé komentáře ap. – srov. cvičení 1 níže. Všechny tyto problémy jsou detailně prodiskutovány např. v [8].

Závěrem je však třeba říci, že i přes zmíněné problémy je konstrukce lexikálního analyzátoru při tvorbě kompilátoru (v porovnání např. s konstrukcí syntaktického analyzátoru) poměrně snadnou záležitostí.

3.2.6 Cvičení

1. Rozšiřte lexikální analyzátor z předcházejícího odstavce 3.2.4 o zpracování komentářů a vynechávání nepotřebných textů (jako ku příkladu “přechod na nový řádek”, nadbytečné mezery apod.).

2. Sestrojte lexikální analyzátor pro identifikátory, které mohou být tvořeny řetězem písmen a číslic (libovolné délky), jenž obsahuje (ovšem ne nutně na prvním místě) nejméně jedno písmeno.

3. Sestrojte analyzátor pro klíčová slova:

BEGIN
FOR
CASE
IF
THEN
ELSE
WHILE
DO
CHAR
END

4. Naprogramujte (např. v PASCALu) lexikální analyzátor programovacího jazyka FORTRAN.

5. Naprogramuje lexikální analyzátor programovacího jazyka PASCAL.

3.3. Syntaktická analýza

Druhou fází překladu je syntaktická analýza. Ve stručnosti jsme se základními metodami syntaktické analýzy zabývali již v odstavci 2.5.3. Nyní si vyložíme princip konstrukce syntaktického analyzátoru podrobněji. Zejména budeme věnovat pozornost deterministickým syntaktickým analyzátorům, které mají pro praxi rozhodující význam.

3.3.1 Definice syntaktické analýzy

Úkolem syntaktického analyzátoru je určit syntaktickou strukturu zdrojového programu. To tedy znamená, že se syntaktický analyzátor snaží sestrojit pro dané slovo lexikálních symbolů (reprezentující zdrojový program) derivační strom. Protože jsou derivační stromy ve vzájemné jednoznačné korespondenci s levými (a pravými) derivacemi, je z praktického hlediska pochopitelně výhodnější hledat pro dané slovo levou (a nebo pravou) derivaci. Syntaktickou analýzu metodou shora dolů resp. zdola nahoru si proto můžeme definovat takto:

DEF

Definice 3.1.1.
Buď = (, , ,) bezkontextová gramatika, která má *n* pravidel očíslovaných 1, ..., *n* a nechť ∈ ( ). Syntaktická analýza metodou shora dolů je proces, který vede k nalezení posloupnosti čísel pravidel užitých při levé derivaci věty . Syntaktická analýza metodou zdola nahoru je proces, který vede k nalezení obrácené posloupnosti čísel pravidel použitých při pravé derivaci věty .

V dalším textu již budeme automaticky předpokládat, že daná gramatika s *n* pravidly (pro nějaké $n \geq 1$) má tato pravidla očíslována čísly 1, …, *n*.

S principem syntaktické analýzy shora dolů i zdola nahoru jsme se předběžně seznámili v odstavci 2.5. Proto již víme, že v prvém případě vlastně provádíme konstrukci derivačního stromu od kořene dolů k větě (což je totéž jako hledat posloupnost čísel pravidel užitých při levé derivaci této věty), zatímco v druhém případě se naopak snažíme danou větu redukovat nahoru ke kořenu (což zase vlastně znamená nalézt obrácenou posloupnost pravidel použitých v pravé derivaci věty). Posloupnost čísel pravidel užitých při levé derivaci věty resp. obrácenou posloupnost čísel pravidel užitých při pravé derivaci věty budeme nazývat levý resp. pravý rozbor této věty. Máme-li tedy problém syntaktické analýzy úspěšně zvládnout, musíme nalézt vhodné prostředky, které překládají věty daného (bezkontextového) jazyka na jejich levé (v případě syntaktické analýzy shora dolů) a nebo pravé (v případě syntaktické analýzy zdola nahoru) rozbory. Studiu těchto prostředků jsou věnovány následující odstavce.

3.3.2. Jednoduchá syntaxí řízená překladová schémata

Prvým formálním prostředkem, pomocí kterého lze jednoduchým způsobem (jak poznáme níže) přeložit věty daného bezkontextového jazyka na jím odpovídající levé resp. pravé rozbory, bude jistý speciální případ syntaxí řízeného překladového schématu:

DEF

Definice 3.3.2.

Buď = (, , $_O$, ,) syntaxí řízené překladové schéma a nechť v každém pravidle $\rightarrow x$, $\in$ jsou přidružené neterminální symboly ve stejném pořadí ve slově x i ve slově . Pak nazýváme jednoduché syntaxí řízené překladové schéma. Překlad) definovaný jednoduchým syntaxí řízeným překladovým schématem se nazývá jednoduchý syntaxí řízený překlad.

Později poznáme, že jednoduchá syntaxí řízená překladová schémata jsou pro nás velmi důležitá. Každý jednoduchý syntaxí řízený překlad lze totiž definovat vhodným zásobníkovým převodníkem, od kterého lze již lehce přejít k realizaci skutečného syntaktického analyzátoru.

x+y

Příklad 3.3.3.

Představme si, že máme za úkol sestrojit jednoduché syntaxí řízené překladové schéma, které překládá (zjednodušené) aritmetické výrazy z jazyka ( ) na aritmetické výrazy bez zbytečných závorek, přičemž bezkontextová gramatika je definována takto:

$$= (\{E, , A\}, \{+, *, (,)\}, Q, E),$$

Kde množina pravidel Q obsahuje těchto osm pravidel:

(1) $E \rightarrow (E)$
(2) $E \rightarrow E + E$
(3) $E \rightarrow$
(4) $\rightarrow ( )$
(5) $\rightarrow A * A$
(6) $\rightarrow$
(7) $A \rightarrow (E + E)$
(8) $A \rightarrow$

Náš úkol lze vyřešit (ku příkladu) pomocí jednoduchého syntaxí řízeného překladového schématu

$$= (\{E, , A\}, \{+, *, (,)\}, \{+, *, (,)\}, , E),$$

kde obsahuje pravidla (která si také očíslujeme):

(1) $E \rightarrow (E), E$
(2) $E \rightarrow E + E, E + E$
(3) $E \rightarrow$,
(4) $\rightarrow ( )$,
(5) $\rightarrow A * A, A * A$
(6) $\rightarrow$,

(7) $A \rightarrow (E + E), (E + E)$
(8) $A \rightarrow$,

Nyní se můžeme například přesvědčit, že věta (((∗))∗)

$(Xx) = \quad (X) \quad (x),$
$_O(Xx) = \quad _O(X) \quad _O(x),$

pro $X \in \quad \cup \quad \cup \quad _O, x \in (\quad \cup \quad \cup \quad _O)^*$.

Pomocí homomorfismu a $_O$ nyní můžeme specifikovat překlad) definovaný překladovou gramatikou .

DEF

Definice 3.3.5.
Překlad) definovaný překladovou gramatikou

$$= (\quad \cup \quad _O, Q,)$$

je definován

$$) = \{(\quad (\quad), \quad _O(\quad)) : \quad \in \quad (\quad)\},$$

kde ( ) je jazyk generovaný .

Buďte $\in (\quad), x \in \quad (\quad), \quad \in \quad _O(\quad)$. Pak x resp. se nazývají vstupní resp. výstupní řetěz popř. věta (a nebo jednoduše: vstup resp. výstup) a se nazývá charakteristickou větou dvojice $(x, \quad)$.

Nyní se budeme zabývat pojmem jednoznačný překlad, o kterém jsme se stručně zmínili již na konci předcházejícího odstavce. Jednoznačný překlad

$$\quad \cup \quad ^* \quad ^*_O$$

je takový překlad, ve které libovolné vstupní větě x odpovídá právě jedna výstupní věta , což zapisujeme takto: $= x)$.

V praxi vždy požadujeme, aby překlad byl jednoznačný, neboť pochopitelně nemůžeme připustit, aby se nám jedna vstupní věta překládala na několik, vzájemně různých vět výstupních.

Překladová gramatika, která definuje jednoznačný překlad, se nazývá jednoznačná.

V souvislosti s překladovou gramatikou se zavádí i pojem tzv. sémantické jednoznačnosti. Překladová gramatika $= (\quad \cup \quad _O, Q,)$ se nazývá sémanticky jednoznačná, právě když platí:

Jestliže $A \to x, A \to \quad \in Q, x \neq \quad ,$
pak $(x) \neq \quad ( )$.

Dohodněme se, že se nadále budeme zabývat pouze překladovými gramatikami, které jsou sémanticky jednoznačné.

DEF

Definice 3.3.6.
Buď

$$= (\quad \cup \quad_O, Q,)$$

překladová gramatika. Pak bezkontextovou gramatikou definovanou

$$= (\quad , Q ,)$$
$$Q = \{A \rightarrow \quad (x): \ A \rightarrow x \in Q \}$$

nazveme vstupní gramatika (překladové gramatiky) a gramatikou $_O$ definovanou

$$_O = (\quad , \quad_O, Q_O,)$$
$$Q_O = \{A \rightarrow \quad_O(x): \ A \rightarrow x \in Q \}$$

nazveme výstupní gramatika (překladové gramatiky).

DEF

Definice 3.3.7.
Dvě překladové gramatiky $_1$, $_2$ jsou ekvivalentní, jestliže

$$_1) = _2)$$

3.3.4. Vzájemný vztah jednoduchých syntaxí řízených překladových schémat a překladových gramatik

Ve srovnání s jednoduchými syntaxí řízenými překladovými schématy jsou překladové gramatiky zřejmě jednodušším a elegantnějším prostředkem pro definici překladu. To je také hlavní důvod, proč jsou zavedeny. Jinak totiž mezi těmito prostředky existují velmi těsné souvislosti, jak si nyní ukážeme.

Věta 3.3.8.
Každý jednoduchý syntaxí řízený překlad lze definovat vhodnou překladovou gramatikou.

Důkaz. Buď

$$= (\quad \quad_O, ,)$$

jednoduché syntaxí řízené překladové schéma. Definujme překladovou gramatiku

$$= (\quad \cup \quad_O, Q,)$$

tak, že

$$) =)$$

Nechť

$$Q = \{A \to x_0\ {}_0B_1x_1\ {}_1 \ldots B_nx_n\ {}_n :$$
$$A \to x_0B_1x_1 \ldots B_nx_n,\ {}_0B_1\ {}_1 \ldots B_n\ {}_n \in$$
$$x_i \in\ ^*,\ {}_i \in\ {}_O^*\ \text{pro}\ 0 \le i \le n$$
$$B_j \in\ \ \text{pro}\ 1 \le j \le n\}$$

Skutečnost, že

$$) =)$$

lze dokázat poměrně jednoduše pomocí indukce podle délky derivace, což je ponecháno na čtenáři.

I obrácené tvrzení věty 3.3.8 platí:

Věta 3.3.9.
Buď překladová gramatika. Pak) je jednoduchý syntaxí řízený překlad.

Důkaz toho tvrzení opět ponecháváme na čtenáři.

Nyní si ilustrujme transformaci jednoduchého syntaxí řízeného schématu na ekvivalentní (tj. stejný překlad definující) překladovou gramatiku (viz Důkaz věty 3.3.8) příkladem.

x+y

Příklad 3.3.10
Uvažujme jednoduché syntaxí řízené překladové schéma, které nám definuje překlad (zjednodušených) aritmetických výrazů z infixové do postfixové polské notace (kterou poznáme později, viz kapitola 3.4). Toto schéma je definováno:

$$= (\{E, B, F\}, \{+, *, (,)\ \ \}, \{+, *,\ \ \}, , E),$$

Kde obsahuje pravidla :

$E \to E + B, EB+$
$E \to B, B$
$B \to B * F, BF*$
$B \to F, F$
$F \to (E), E$
$F \to\ \ ,\ \ .$

Jak víme, v překladové gramatice musí být vstupní a výstupní abeceda disjunktní, což ovšem naše jednoduché syntaxí řízené překladové schéma nesplňuje. Proto si jednoduchým způsobem vstupní a výstupní symboly rozlišíme, např. si všechny symboly ve vstupní abecedě zakroužkujeme. Výstupní abeceda se tedy rovná $\{\oplus, \otimes,\ \ \}$. Nyní již můžeme provést

transformaci schématu na překladovou gramatiku dle důkazu věty 3.3.8. Získáváme překladovou gramatiku:

$$= (\{E, B, F\}, \{+, *, (,), \ \} \cup \{\oplus, \otimes, @\}, Q, E),$$

kde Q obsahuje pravidla:

$E \rightarrow E + B \oplus$
$E \rightarrow B$
$B \rightarrow B * F \otimes$
$B \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow \ @$

Na této gramatice si nyní ilustrujeme některé pojmy definované v předcházejícím odstavci 3.3.3. Uvažujme např. odvození:

$E \Rightarrow B$
$\Rightarrow B$
$\Rightarrow B * F$
$\Rightarrow F * F$
$\Rightarrow \ @ * (E) \otimes$
$\Rightarrow \ @ * (E + B \oplus) \otimes$
$\Rightarrow \ @ * (B + B \oplus) \otimes$
$\Rightarrow \ @ * (F + B \oplus) \otimes$
$\Rightarrow \ @ * (\ @ + B \oplus) \otimes$
$\Rightarrow \ @ * (\ @ + F \oplus) \otimes$
$\Rightarrow \ @ * (\ @ + \ @ \oplus) \otimes,$

což tedy znamená, že

$$* (\quad + \quad \oplus) \otimes \in \ (\ )$$

je charakteristická věta pro vstupní větu:

$$* (\ + \ );$$

Výstupní větou je řetěz:

$$\oplus \otimes$$

3.3.5. Zásobníkové převodníky

Zásobníkový převodník je v podstatě zásobníkový automat (srov. Definici 2.5.2), který má však navíc v každém taktu možnost výstupu řetězu nad výstupní abecedou.

Formálně:

DEF

Definice 3.3.11.

Zásobníkový převodník je osmice

$$= (\ ,\ ,\ Z,\ _O,\ \delta,\ q_0,\ Z_0,\ F),$$

kde

- konečná množina stavů;
- vstupní abeceda;
- Z - zásobníková abeceda;
- $_O$ - výstupní abeceda;
- δ - zobrazení $\times$ ($\cup \{\varepsilon\}$) $\times Z$ do množiny konečných podmnožin množiny $\times Z^* \times\ _O^*$;
- $q_0 \in$ - počáteční stav;
- $Z_0 \in Z$ - počáteční symbol zásobníku;
- $F \subseteq$ - množina koncových stavů.

Konfigurace zásobníkového převodníku je čtveřice

$$(q,\ x,\ ,\ )$$

kde

- $q \in$ je aktuální stav ;
- $x \in\ ^*$ je sufix vstupního řetězu, jenž doposud nebyl přečten;
- $\in Z^*$ je stávající obsah zásobníku;
- $\in\ _O^*$ je doposud vytvořený prefix výstupního řetězu.

Podobně, jako pro zásobníkové automaty, i pro zásobníkové převodníky definujeme (na množině konfigurací) relaci $\mapsto$ (takt):

Jestliže $(r,\ ,\ ) \in \delta(q,\ ,\ B)$, kde
$r, q \in\ ,\ \cup \{\varepsilon\} \in\ ,\ B \in Z,$
$\in Z^*,\ \in\ _O^*$
a $x \in\ ^*,\ \in Z^*,\ \in\ _O^*$
pak $(q,\ x, B\ ,\ ) \mapsto (r, x,\ ,\ )$.

Analogicky jako pro zásobníkové automaty můžeme i pro zásobníkové převodníky definovat tranzitivní a reflexivní uzávěr $\mapsto^*$ relace $\mapsto$. Řekněme, že $\in\ _O^*$ je výstupem pro $x \in\ ^*$ jestliže

$$(q_0, x, Z_0, \varepsilon) \mapsto^* (q, \varepsilon,\ ,\ )$$

pro nějaké $q \in F,\ \in Z^*$; $(q_0, x, Z_0, \varepsilon)$ resp. $(q, \varepsilon,\ ,\ )$ se nazývá počáteční resp. koncová konfigurace.

Překlad) definovaný zásobníkovým převodníkem je množina

$$) = \{(x,): (q_0, x, Z_0, \varepsilon) \mapsto^* (q, \varepsilon, ,) \text{ pro nějaké } q \in , \in Z^*\}.$$

Překlad ( ) definovaný zásobníkovým převodníkem s prázdným zásobníkem je množina dvojic:

$$( ) = \{(x,): (q_0, x, Z_0, \varepsilon) \mapsto^* (q, \varepsilon, \varepsilon,) \text{ pro nějaké } q \in \}.$$

Podobně jako lze pro zásobníkové automaty dokázat větu 2.5.22, můžeme pro zásobníkové převodníky dokázat:

Věta 3.3.12.
Překlad = (1) pro nějaký zásobníkový převodník 1, právě když = 2), pro vhodný zásobníkový převodník 2.

Důkaz je analogický jako důkaz věty 2.5.22, a proto jej ponecháme na čtenáři.

V praxi je pochopitelně vždy výhodné, když je modelem syntaktického analyzátoru deterministický zásobníkový převodník. Proto si jej nyní definujme:

DEF

Definice 3.3.13.
Zásobníkový převodník

$$= (, Z, O, \delta, Z_0, F)$$

se nazývá deterministický, jestliže:

(1) pro každé $q \in$, $\in$ $\cup \{\varepsilon\}$, $B \in Z$ množina $\delta(q, , B)$ obsahuje nejvýše jeden prvek,
(2) pro každé $q \in$, $B \in Z$, $\in$ platí: jestliže $\delta(q, \varepsilon, Z) \neq 0$, pak $\delta(q, , Z) = 0$.

Ilustrujme si pojem deterministického zásobníkového převodníku jedním zajímavým příkladem.

x+y

Příklad 3.3.14.
V kapitole 3.4 poznáme (prefixovou a postfixovou) polskou notaci. Předběžně můžeme říci, že se jedná o zápis, pomocí kterého vytváříme vnitřní formu programu.

V tomto příkladu si uvedeme deterministický zásobníkový převodník, který překládá (zjednodušené) aritmetické výrazy v prefixové polské notaci na aritmetické výrazy v postfixové polské notaci.
Tento převodník je definován takto:

$$= (\{q\}, \{ , +, *\}, \{ , *, E\}, \{+, *, \}, \delta, q, E, \{q\}),$$

kde δ je definováno takto:

$\delta(q, \ , E) = \{(q, \varepsilon, \ )\}$,
$\delta(q, +, E) = \{(q, EE+, \varepsilon)\}$,
$\delta(q, *, E) = \{(q, EE*, \varepsilon)\}$,
$\delta(q, \varepsilon, +) = \{(q, \varepsilon, +)\}$,
$\delta(q, \varepsilon, *) = \{(q, \varepsilon, *)\}$.

Vstup

$+*$

přeloží tento deterministický zásobníkový převodník na výstup

$* \ +$

takto:

$$\begin{aligned}(q, +* \ , E, \varepsilon) &\mapsto (q, * \ , EE+, \varepsilon)\\ &\mapsto (q, \ , EE*E+, \varepsilon)\\ &\mapsto (q, \ , E*E+, \ )\\ &\mapsto (q, \ , *E+, \ )\\ &\mapsto (q, \ , E+, \ *)\\ &\mapsto (q, \varepsilon, +, \ * \ )\\ &\mapsto (q, \varepsilon, \varepsilon, \ * \ +).\end{aligned}$$

Tento deterministický zásobníkový převodník tedy definuje překlad

$\ ) = \{(x, \ ):$ x je zápis aritmetického výrazu v prefixové polské notaci, je zápis téhož aritmetického výrazu v postfixové polské notaci$\}$.

Viděli jsme, že zásobníkový převodník získáme velmi jednoduchým rozšířením zásobníkového automatu. Analogicky získáme z rozšířeného zásobníkového automatu rozšířený zásobníkový převodník. Dohodněme se však, že vrchol zásobníku rozšířeného zásobníkového převodníku bude vpravo (a nikoliv vlevo, jako tomu bylo v případě zásobníkového automatu z definice 3.3.11).

DEF

Definice 3.3.15.
Rozšířený zásobníkový převodník je osmice

$$= (\ , Z, \ _O, \delta, q_0, Z_0, F),$$

kde všechny symboly mají stejný význam jako v definici 3.3.11 až na to, že δ nyní označuje zobrazení konečné podmnožiny množiny $\times (\cup \{\varepsilon\}) \times Z^*$ do množiny konečných podmnožin množiny $\times Z^* \times \ _O{}^*$. Píšeme $(q, \ , \ , x) \mapsto (p, \ , \ , x \ )$, jestliže $(p, \ ) \in \delta(q, \ , \ )$, $p, q \in \ , \ \in \cup \{\varepsilon\}, \ , \ , \in Z^*, \ \in \ ^*, x, \ \in \ _O{}^*$.

DEF

Definice 3.3.16.
Rozšířený zásobníkový převodník

$$= (\ \ , Z, \ _O, \delta, q_0, Z_0, F)$$

je deterministický, jestliže

(1) $\delta(q, \ , \ )$ obsahuje nejvýše jeden prvek pro libovolné $q \in \ , \ \in \ \cup \{\varepsilon\}, x \in Z^*$;
(2) Slova $\ , \ u$ nejsou sufixem jeden druhému, jestliže $\delta(q, \ , \ ) \neq 0$ a $\delta(q, \ , \ u) \neq 0$ pro $\ = \$ nebo $\ = \varepsilon$.

3.3.6. Vzájemný vztah jednoduchých syntaxí řízených překladových schémat a zásobníkových převodníků

V tomto odstavci ukážeme, že jednoduchá syntaxí řízená překladová schémata a zásobníkové převodníky jsou ekvivalentní prostředky pro definici překladu.

Lemma 3.3.17.
Buď $= (\ , \ , \ _O, \ , \ )$ jednoduché syntaxí řízené překladové schéma. Pak existuje zásobníkový převodník $_2$ tak, že $\ _1) = \ (\ _2)$.

Důkaz. Mějme jednoduché syntaxí řízené překladové schéma

$$_1 = (\ , \ , \ _O, \ , \ ) \text{ a}$$

(bez újmy na obecnosti) předpokládejme: $\ \cap \ _O = \varnothing$. Sestrojme zásobníkový převodník

$$_2 = (\{q\}, \ , \ \cup \ \cup \ _O, \ _O, \delta, q, \ , \varnothing),$$

kde δ je definováno takto:

(1) jestliže
$A \to x_0 B_1 x_1 \ldots B \ x \ , \ _0 B_1 \ _1 \ldots B \ \in \ ,$
$x_i \in \ ^*, \ _i \in \ _O^*, A, B_j \in \ , 1 \leq j \leq \ , 0 \leq i \leq \ , \ \geq 0,$

pak
$(q, x_0 \ _0 B_1 x_1 y_1 \ldots B \ x \ , \varepsilon) \in \delta(q, \varepsilon, A)$;

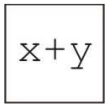
(2) $\delta(q, \ , \ ) = \{(q, \varepsilon, \varepsilon)\}$, pro každé $\in \ ;$
(3) $\delta(q, \varepsilon, \ ) = \{(q, \varepsilon, \ )\}$, pro každé $\in \ _O$.

Důkaz rovnosti

$$\ _1) = \ (\ _2)$$

lze provést indukcí podle délky překladové derivace v $_1$ a počtu taktů v $_2$, což je ponecháno na čtenáři, viz též [15].

Použití konstrukce presentované ve výše uvedeném důkazu si nyní ilustrujme příkladem.



Příklad 3.3.18.

Uvažujme jednoduché syntaxí řízené překladové schéma z příkladu 3.3.10. Zakroužkováním symbolů ve výstupní abecedě (tzn. místo {+, *, } uvažujeme abecedu {⊕, ⊗, @}) si obdobně jako v příkladu 3.3.10 zajistíme, aby vstupní a výstupní abeceda byly disjunktní. Zásobníkový převodník ′, který sestrojíme podle konstrukce presentované v důkazu lemmatu 3.3.17 a jenž nám tedy definuje překlad (′) =), bude definován takto:

$$
\begin{aligned}
&{}' = (\{q\}, \{+, *, (,), \ \}, \{+, *, (,), \ , E, B, F, \oplus, \otimes, @\},\\
&\qquad \{\oplus, \otimes, @\}, \delta, q, E, \varnothing),\\
&\delta(q, \varepsilon, E) = \{(q, E + B \oplus, \varepsilon), (q, B, \varepsilon)\},\\
&\delta(q, \varepsilon, B) = \{(q, B * F \otimes, \varepsilon), (q, F, \varepsilon)\},\\
&\delta(q, \varepsilon, F) = \{(q, (E), \varepsilon), (q, \ @, \varepsilon)\},\\
&\delta(q, c, c) = \{(q, \varepsilon, \varepsilon)\} \text{ pro každé } c \in \{+, *, (,), \ \},\\
&\delta(q, \varepsilon, \ ) = \{(q, \varepsilon, \ )\} \text{ pro každé } \ \in \{\oplus, \otimes, @\}.
\end{aligned}
$$

Podobně jako v příkladu 3.3.10 uvažujme i nyní vstupní větu

$$* (\ + \ )$$

a podívejme se na posloupnost taktů, kterou zásobníkový převodník ′ pro tuto větu provede (vrchol zásobníku je vlevo):

$$
\begin{aligned}
(q, \ * (\ + \ ), E, \varepsilon) &\mapsto (q, \ * (\ + \ ), B, \varepsilon)\\
&\mapsto (q, \ * (\ + \ ), B * F \otimes, \varepsilon)\\
&\mapsto (q, \ * (\ + \ ), F * F \otimes, \varepsilon)\\
&\mapsto (q, \ * (\ + \ ), \ @ * F \otimes, \varepsilon)\\
&\mapsto (q, * (\ + \ ), @ * F \otimes, \varepsilon)\\
&\mapsto (q, * (\ + \ ), * F \otimes, @)\\
&\mapsto (q, (\ + \ ), F \otimes, @)\\
&\mapsto (q, (\ + \ ), (E) \otimes, @)\\
&\mapsto (q, \ + \ ), E) \otimes, @)\\
&\mapsto (q, \ + \ ), E + B \oplus) \otimes, @)\\
&\mapsto (q, \ + \ ), B + B \oplus) \otimes, @)\\
&\mapsto (q, \ + \ ), F + B \oplus) \otimes, @)\\
&\mapsto (q, \ + \ ), \ @ + B \oplus) \otimes, @)\\
&\mapsto (q, + \ ), @ + B \oplus) \otimes, @)\\
&\mapsto (q, + \ ), + B \oplus) \otimes, @ \ @)\\
&\mapsto (q, \ ), B \oplus) \otimes, @ \ @)\\
&\mapsto (q, \ ), F \oplus) \otimes, @ \ @)\\
&\mapsto (q, \ ), \ @ \oplus) \otimes, @ \ @)\\
&\mapsto (q,), @ \oplus) \otimes, @ \ @)
\end{aligned}
$$

$\mapsto (q,), \oplus) \otimes, @\ @\ @)$
$\mapsto ((q,),) \otimes, @\ @\ @\ \oplus)$
$\mapsto (q, \varepsilon, \otimes, @\ @\ @\ \oplus)$
$\mapsto (q, \varepsilon, \varepsilon, @\ @\ @\ \oplus\ \otimes).$

Lemma 3.3.19.
Buď = (Q, , Z, $_O$, δ, q_0, Z_0, F) zásobníkový převodník, pak ( ) je jednoduchý syntaxí řízený překlad.

Důkaz: Viz [15].

3.3.7. Syntaktická analýza shora dolů

Výklad syntaktické analýzy metodou shora dolů začneme zcela obecně. Ukážeme si, jak lze pro libovolný bezkontextový jazyk sestrojit zásobníkový převodník, který věty tohoto jazyka překládá na jejich levé rozbory. Tento zásobníkový převodník (a tedy model syntaktického analyzátoru) je však (v obecném případě) nedeterministický. Nicméně, v následujícím odstavci si ukážeme, že pro určité speciální podtřídy bezkontextových jazyků lze tento nedeterminismus odstranit. Nejprve si pro daný bezkontextový jazyk sestrojíme jednoduché syntaxí řízené překladové schéma, které provádí jeho syntaktickou analýzu shora dolů.

Věta 3.3.22.
Buď bezkontextová gramatika. Pak existuje jednoduché syntaxí řízené překladové schéma tak, že

$$) = \{(x,): x \in ( ), \text{ je levý rozbor } x\}.$$

Důkaz. Buď

$$= (, ,)$$

bezkontextová gramatika s n pravidly (pro nějaké $n \geq 1$) očíslovanými 1, ..., n.

Definujme si jednoduché syntaxí řízené překladové schéma

$$= (, , \{1, ..., n\}, ,),$$

kde pravidla z jsou definována takto:

Jestliže $A \rightarrow x \in$ je pravidlo s číslem l, $l \in \{1, ..., n\}$, $A \in$, $x \in (\cup)^*$, pak $A \rightarrow x, l \in$, kde je slovo x, z kterého jsou eliminovány všechny terminály.

Důkaz, že vskutku

$$) = \{(x,): x \in ( ), \text{ je levý rozbor } x\}$$

lze provést velmi snadno pomocí indukce, což je ponecháno na čtenáři (viz též [2]).

Příklad 3.3.23.
Buď

$$= (\{E, B, F\}, \{\ \ , *, +, (,)\},\ E)$$

bezkontextová gramatika a nechť obsahuje pravidla

(1) $E \to E + B$
(2) $E \to B$
(3) $B \to B * F$
(4) $B \to F$
(5) $F \to (E)$
(6) $F \to\ \ .$

Všimněme si, že například větu

$$*\ (\ \ +\ \ )$$

lze v odvodit levou derivací (vpravo uvádíme použitá pravidla):

$$\begin{array}{lll} E & \Rightarrow & B \qquad\qquad\quad (2) \\ & \Rightarrow & B * F \qquad\quad (3) \\ & \Rightarrow & F * F \qquad\quad (4) \\ & \Rightarrow & \ \ * F \qquad\quad (6) \\ & \Rightarrow & \ \ * (E) \qquad\quad (5) \\ & \Rightarrow & \ \ * (E + B) \qquad (1) \\ & \Rightarrow & \ \ * (B + B) \qquad (2) \\ & \Rightarrow & \ \ * (F + B) \qquad (4) \\ & \Rightarrow & \ \ * (\ \ + B) \qquad (6) \\ & \Rightarrow & \ \ * (\ \ + F) \qquad (4) \\ & \Rightarrow & \ \ * (\ \ +\ \ ) \qquad (6) \end{array}$$

a tedy, že její levý rozbor je

$$2\ 3\ 4\ 6\ 5\ 1\ 2\ 4\ 6\ 4\ 6.$$

Podle důkazu věty 3.3.22 sestrojíme jednoduché syntaxí řízené překladové schéma

$$= (\{E, B, F\}, \{+, *, (,),\ \ \}, \{1, ..., 6\}, , E),$$

kde obsahuje pravidla

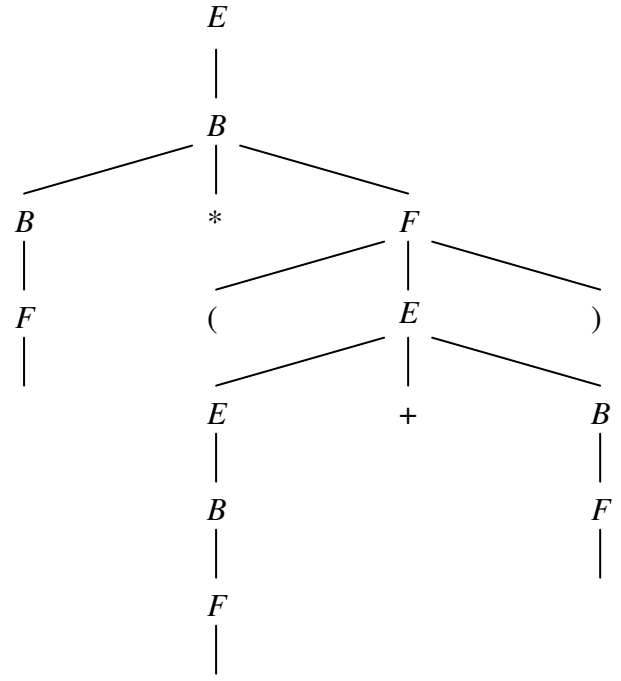
$$E \to E + B,\ 1\ E\ B$$

$E \rightarrow B$, 2 B
$B \rightarrow B$ * F, 3 B F
$B \rightarrow F$, 4 F
$E \rightarrow (E)$, 5 E
$F \rightarrow$, 6

Nyní si ukažme, jak v vypadá derivace, která určí výstup (tj. levý rozbor) vstupního řetězu * (+):

$$
\begin{aligned}
(E, E) &\Rightarrow (B, 2\ B) \\
&\Rightarrow (B + F, 2\ 3\ B\ F) \\
&\Rightarrow (F * F, 2\ 3\ 4\ F\ F) \\
&\Rightarrow (\ * F, 2\ 3\ 4\ 6\ F) \\
&\Rightarrow (\ * (E), 2\ 3\ 4\ 6\ 5\ E) \\
&\Rightarrow (\ * (E + B), 2\ 3\ 4\ 6\ 5\ 1\ E\ B) \\
&\Rightarrow (\ * (B + B), 2\ 3\ 4\ 6\ 5\ 1\ 2\ B\ B) \\
&\Rightarrow (\ * (F + B), 2\ 3\ 4\ 6\ 5\ 1\ 2\ 4\ F\ B) \\
&\Rightarrow (\ * (\ + B), 2\ 3\ 4\ 6\ 5\ 1\ 2\ 4\ 6\ B) \\
&\Rightarrow (\ * (\ + F), 2\ 3\ 4\ 6\ 5\ 1\ 2\ 4\ 6\ 4\ F) \\
&\Rightarrow (\ * (\ + \ ), 2\ 3\ 4\ 6\ 5\ 1\ 2\ 4\ 6\ 4\ 6);
\end{aligned}
$$

tedy derivačnímu stromu



pro větu * (+) ve vstupní gramatice schématu odpovídá derivační strom

ve výstupní gramatice tohoto schématu .

Podle věty 3.3.21 jsou jednoduchá syntaxí řízená překladová schémata a zásobníkové převodníky ekvivalentní (tj. stejnou třídu překladů definující) prostředky. Ukažme si tedy nyní, jak lze syntaktickou analýzu shora dolů daného bezkontextového jazyka provést pomocí zásobníkového převodníku.

Věta 3.3.24.

Buď bezkontextová gramatika. Pak existuje zásobníkový převodník takový, že

( ) = {(,): $x \in$ ( ), je levý rozbor x}.

Důkaz. Buď

= (, ,)

bezkontextová gramatika s pravidly jako obvykle očíslovanými od 1 do n (pro nějaké $n \geq 1$). Sestrojme (nedeterministický) zásobníkový převodník takto:

= ({q}, , $\cup$, {1, ..., n}, δ, q, , Φ),

kde δ definujeme:

(1) jestliže $A \rightarrow x \in$ je pravidlo s číslem i, $i \in$ {1, ..., n},
pak (q, x, i) $\in$ δ(q, ε, A);
(2) jestliže di

Tím je konstrukce dokončena. Formální dokončení důkazu, tzn. že

$$\mathcal{L}(\quad) = \{(x, \quad): x \in \quad (\quad), \quad \text{je levý rozbor } x\}$$

je ponechán na čtenáři, viz též [??].

DEF

Definice 3.3.25.

Zásobníkový převodník sestrojený v důkazu věty 3.3.24 se nazývá levý analyzátor.

Podobně jako jsme si v příkladu 3.3.23 předvedli, jak se provádí syntaktická analýza metodou shora dolů pomocí jednoduchého syntaxí řízeného překladového schématu, ukážeme si v následujícím příkladu, jak provádí syntaktickou analýzu shora dolů levý analyzátor.

x+y

Příklad 3.3.26.

Sestrojíme levý analyzátor pro gramatiku z příkladu 3.3.23:

$$= (\{q\}, \{\ , *, +, (,)\}, \{E, B, F, \ , *, +, (,)\}, \{1, ..., 6\}, \delta, q, E, \varnothing),$$

$$\delta(q, \varepsilon, E) = \{(q, E + B, 1), (q, B, 2)\},$$
$$\delta(q, \varepsilon, B) = \{(q, B * F, 3), (q, F, 4)\},$$
$$\delta(q, \varepsilon, F) = \{(q, (E), 5), (q, \ , 6)\},$$
$$\delta(q, \natural, \natural) = \{(q, \varepsilon, \varepsilon)\} \text{ pro každé } \natural \in \{\ , +, *, (,)\}.$$

Uvažujme větu

$$+ \quad * \quad \in \quad (\quad)$$

a ukažme si, jak levý analyzátor provede její syntaktickou analýzu:

$$\begin{aligned}
(q, \quad + \quad * \quad , E, \varepsilon) &\vdash (q, \quad + \quad * \quad , E + B, 1) \\
&\vdash (q, \quad + \quad * \quad , B + B, 12) \\
&\vdash (q, \quad + \quad * \quad , F + B, 124) \\
&\vdash (q, \quad + \quad * \quad , \quad + B, 1246) \\
&\vdash (q, + \quad * \quad , + B, 1246) \\
&\vdash (q, \quad * \quad , B, 1246) \\
&\vdash (q, \quad * \quad , B * F, 12463) \\
&\vdash (q, \quad * \quad , F * F, 124634) \\
&\vdash (q, \quad * \quad , \quad * F, 1246346) \\
&\vdash (q, * \quad , * F, 1246346) \\
&\vdash (q, \quad , F, 1246346) \\
&\vdash (q, \quad , \quad , 12463466) \\
&\vdash (q, \varepsilon, \varepsilon, 12463466).
\end{aligned}$$

3.3.8. LL gramatiky a deterministická syntaktická analýza shora dolů

Z praktického hlediska je krajně nepříjemná skutečnost, že levý analyzátor sestrojený v důkazu věty 3.3.24 je obecně nedeterministický. Nadto již z odstavce 2.4 víme, že existují bezkontextové jazyky, jež nelze akceptovat žádným deterministickým zásobníkovým automatem. Pro takovéto jazyky pak ale pochopitelně také není možné nalézt deterministický zásobníkový převodník, který by provedl jejich syntaktickou analýzu shora dolů.

Nicméně, existuje speciální typ bezkontextových gramatik, pro které lze vždy nalézt deterministický syntaktický analyzátor pracující metodou shora dolů. Jedná se o tzv. LL gramatiky, kterým budeme (v souvislosti se syntaktickou analýzou shora dolů) věnovat pozornost v tomto odstavci. Celý výklad zahájíme jednoduchým příkladem, ve kterém se pokusíme ukázat, jak nám čtené symboly v některých případech mohou jednoznačně specifikovat takt, který má zásobníkový převodník provést a tím nám umožnit jeho deterministickou konstrukci. Tato skutečnost ostatně tvoří (jak hned poznáme) základní princip zavedení a zkoumání LL gramatik.

x+y

Příklad 3.3.27.

Mějme bezkontextovou gramatiku

$$= (\{\blacksquare\}, \{\ , \flat\ , c\}, \blacksquare, \blacksquare),$$

kde P obsahuje tři pravidla (která si opět pro jednoduchost očíslujeme):

(1) $\blacksquare \rightarrow \ \blacksquare$
(2) $\blacksquare \rightarrow \flat\ \blacksquare\flat$
(3) $\blacksquare \rightarrow c$.

Tato gramatika generuje jazyk

$$(\ \ ) = \{xcx^{\mathrm{R}}: x \in \{\ , \flat\}^*\}.$$

Pomocí metody uvedené v důkazu věty 3.3.24 sestrojme zásobníkový převodník (tj. levý analyzátor):

$$_1 = (\{q\}, \{\ , \flat\ , c\}, \{\ , \flat\ , c, \blacksquare\}, \{1, 2, 3\}, \delta, q, \blacksquare, \varnothing),$$
$$\delta(q, \varepsilon, \blacksquare) = (q, \ \blacksquare\ , 1),$$
$$\delta(q, \varepsilon, \blacksquare) = (q, \flat\blacksquare\flat, 2),$$
$$\delta(q, \varepsilon, \blacksquare) = (q, c, 3),$$
$$\delta(q, \ , \ ) = (q, \varepsilon, \varepsilon),$$
$$\delta(q, \flat, \flat) = (q, \varepsilon, \varepsilon),$$
$$\delta(q, c, c) = (q, \varepsilon, \varepsilon);$$

tak, že

$$(\ _1) = \{(x, \ ): x \in \ (\ ), \quad \text{je levý rozbor } x\}.$$

Podívejme se, jak vypadá např. překlad slova

$$\flat c\flat \ \in \ (\ )$$

pomocí $_1$:

$$\begin{aligned}(q,\ \flat c\flat\ ,\ \aleph,\ \varepsilon) &\vdash (q,\ \flat c\flat\ ,\ \aleph\ ,\ 1)\\ &\vdash (q, \flat c\flat\ ,\ \aleph\ ,\ 1)\\ &\vdash (q, \flat c\flat\ ,\ \flat\ \aleph\ ,\ 12)\\ &\vdash (q,\ c\flat\ ,\ \aleph\ ,\ 12)\\ &\vdash (q,\ c\flat\ ,\ c\flat\ ,\ 123)\\ &\vdash (q, \flat\ ,\ \flat\ ,\ 123)\\ &\vdash (q,\ \ ,\ \ ,\ 123)\\ &\vdash (q, \varepsilon, \varepsilon, 123)\end{aligned}$$

tj. levý rozbor věty

$$\flat c\flat\ \in\ (\ ) \text{ je } 123$$

neboť

$$(\ \flat c\flat\ , 123)\ \in\ \_{1}).$$

Zásobníkový převodník $_1$ není deterministický. Všimněme si, ale, že čteného symbolu lze využít při rozhodování o tom, podle kterého pravidla bude takt převodníku $_1$ proveden. Tak např. při akceptování slova $\flat c\flat$ jsme nutně museli prvý takt provést podle pravidla (1), neboť kdybychom jej provedli podle pravidla (2) resp. (3), pak se nám na vrchol zásobníku dostane symbol $\flat$ resp. c a vzhledem k tomu, že je čtený symbol , nemohl by pak již být proveden další takt. Pokusme se tedy využít čtených symbolů ke konstrukci deterministického zásobníkového převodníku $_2$, pro který by platilo

$$(\ _1) = (\ _2).$$

$$_2 = (\{q\}\ ,\ \{\ ,\flat\ ,\ c\},\ \{\ ,\flat\ ,\ c,\ \aleph\},\ \{1, 2, 3\},\ \delta,\ q,\ \aleph,\ \varnothing)$$

$$\begin{aligned}&\delta(\ ,\ ,\ \aleph) = (q,\ \aleph\ ,\ 1)\\ &\delta(q, \flat\ ,\ \aleph) = (q,\ \flat\ \aleph\ ,\ 2)\\ &\delta(q,\ c,\ \aleph) = (q,\ \varepsilon,\ 3)\\ &\delta(q,\ \ ,\ \ ) = (q,\ \varepsilon,\ 1)\\ &\delta(q, \flat\ ,\ \flat\ ) = (q,\ \varepsilon,\ \varepsilon)\\ &\delta(q,\ c,\ c) = (q,\ \varepsilon,\ \varepsilon)\end{aligned}$$

Překlad slova

$$\flat c\flat\ \in\ (\ )$$

pak ovšem vypadá takto:

$$\begin{aligned}(q,\ \flat c\flat\ ,\ \aleph,\ \varepsilon) &\vdash (q, \flat c\flat\ ,\ \aleph\ ,\ 1)\\ &\vdash (q,\ c\flat\ ,\ \flat\ \aleph\ ,\ 12)\end{aligned}$$

$\vdash (q, \flat\ , \flat\ , 123)$
$\vdash (q,\ ,\ , 123)$
$\vdash (q, \varepsilon, \varepsilon, 123)$

a tedy

$(\ \flat\ c\flat\ , 123) \in \delta(\ _2)$.

Funkci δ si můžeme též znázornit pomocí tzv. rozborové tabulky, kterou si v dalším textu budeme označovat :

vstupní symbol

vrchol zásobníku	δ		$\flat$	c	ε
		, 1	, 2	ε, 3	-
		x	-	-	-
	$\flat$	-	x	-	-
	#	-	-	-	-

Řádky tabulky jsou označeny symboly ze zásobníkové abecedy, sloupce jsou označeny symboly ze vstupní abecedy a na průsečíku příslušného řádku a sloupce je určena operace, kterou má levý analyzátor provést v závislosti na vstupu (ε znamená, že slovo je celé přečteno) a na vrcholu zásobníku (# je speciální symbol uložený na dně zásobníku; objeví-li se na vrcholu zásobníku, znamená to tedy, že je zásobník vyprázdněn).

Existují čtyři typy operací:

(1) Srovnání vstupního symbolu s prvním symbolem na pravé straně i-tého pravidla, na jehož levé straně je neterminál, který je na vrcholu zásobníku. Vrchol zásobníku se nahradí pravou stranou i-tého pravidla, ve kterém je vynechán první (terminální) symbol. Číslo i (aplikovaného pravidla) je zprava zřetězeno k výstupnímu řetězu. Oba údaje (tj. zbytek pravé strany pravidla i jeho číslo) jsou v tabulce uvedeny.

(2) x : Srovnání vrcholu zásobníku a vstupního symbolu. V příštím taktu bude čten následující vstupní symbol.

(3) - : Chyba, tj. vstupní slovo nepatří do ( ).

(4) + : Přijetí, tj. vstupní slovo patří do ( ) a jeho překlad (tj. levý rozbor) je vytvořen.

Všimněme si, že v rozborové tabulce není vůbec uveden stav převodníku. Toto zjednodušení je pochopitelné: podle konstrukce (viz důkaz věty 3.2.24) má levý analyzátor pouze jeden stav, a nemá proto smysl jej v specifikovat.

Pokusme se nyní možnost konstrukce deterministického zásobníkového převodníku (jak se intuitivně ukazovala v předcházejícím příkladě) vyjádřit přesně. Tak především vyvstává zásadní otázka: pro jaký typ bezkontextových gramatik lze vždy vytvořit rozborovou tabulku? Odpověď zní: pro tzv. jednoduché LL(1) gramatiky. Definujme si je:

DEF

Definice 3.3.28.
Bezkontextová gramatika = (, , ,) se nazývá jednoduchá LL(1) gramatika, jestliže splňuje tyto dvě podmínky:

(1) pravá strana libovolného pravidla začíná terminálem;
(2) jestliže dvě pravidla mají stejnou levou stranu, pak se liší v terminálu, kterým začíná jejich pravá strana.

Snadno se přesvědčíme, že bezkontextová gramatika G z příkladu 3.3.27 je jednoduchá LL(1) gramatika.

Existuje algoritmus, pomocí kterého lze pro libovolnou jednoduchou LL(1) gramatiku sestrojit rozborovou tabulku . Uveďme se jej:

Algoritmus 3.3.29.
Vytvoření rozkladové tabulky pro jednoduché LL(1) gramatiku.

Vstup: Jednoduchá LL(1) gramatika = (, , ,).

Výstup: Rozborová tabulka pro gramatiku .

Metoda: je definována na $\cup \bar{} \cup \{\#\} \times \cup \{\varepsilon\}$,

kde $\bar{}$ je podmnožina , obsahující terminály, které se vyskytují na pravé straně některého pravidla jinde než na začátku.

(1) jestliže $A \rightarrow x$ je i-té pravidlo, $\in$, $x \in (\cup)^*$, $A \in$,
pak $(A,) = x, i$;
(2) $(,) = \times$ pro každé $\in \bar{}$;
(3) $(\#, \varepsilon) = +$;
(4) $(B,) = -$ ve všech zbývajících případech.

Právě uvedený algoritmus si nyní ilustrujme příkladem.

x+y

Příklad 3.5.30.
Uvažujeme bezkontextovou gramatiku

$$= (\{ , A\}, \{ , c, \}, ,);$$

kde obsahuje pravidla:

(1) $\mathfrak{S} \rightarrow \mathfrak{S}A$
(2) $\mathfrak{S} \rightarrow \mathfrak{b}Ac$
(3) $A \rightarrow A$
(4) $A \rightarrow c$.

Všimněme si, že množina $\subseteq \{B, c, \}$, mající vlastnosti požadované ve výše uvedeném algoritmu je rovna {c}, tj. je definována na

$$\{\mathfrak{S}, A, c, \#\} \times \{\mathfrak{b}, c, , \varepsilon\}.$$

Užitím algoritmu 3.3.29 získáme tedy rozborovou tabulku :

δ	$\mathfrak{b}$	c		ε
$\mathfrak{S}$	Ac,2	-	$\mathfrak{S}A$,1	-
A	-	ε,4	A,3	-
c	-	×	-	-
#	-	-	-	+

Nyní si ale již můžeme pro jednoduché LL(1) gramatiky sestrojit deterministický levý analyzátor.

Věta 3.3.31.

(Syntaktická analýza shora dolů pro jednoduché LL(l) gramatiky.)

Buď rozborová tabulka pro jednoduchou LL(1) gramatiku = (, , $\mathfrak{S}$), která má svá pravidla očíslována 1, ..., *n*. Pak existuje deterministický levý analyzátor *Z*, který provádí syntaktickou analýzu (shora dolů) jazyka ( ), tj. překlad

$$(Z) = \{(x,): x \in ( ) \text{ a } \text{ je levý rozbor } x\}.$$

Důkaz. Nechť předpoklady našeho tvrzení platí. Definujme zásobníkový převodník

$$Z = (\{q\}, , \cup \cup \{\#\}, \langle \text{CHYBA} \rangle \cup \{1, ..., n\}, \delta, q, \mathfrak{S}, \varnothing),$$

kde má stejný význam jako v algoritmu 3.3.29. δ je definováno takto (srov. body (1) – (4) z algoritmu 3.5.29.):

(1) jestliže $(A,) = x, i$,
pak $\delta(q, , A) = \{(q, x, i)\}$,
(2) jestliže M(a, a) = x,
pak $\delta(q, a, a) = \{(q, e, e)\}$;
(3) jestliže M(#, e) = +,
pak $\delta(q, e, \#) = \{(q, \#, e)\}$;
(4) jestliže M(B, a) = -,
pak $\delta(q, e, B) = \{(q, e, < \text{CHYBA} >)\}$.

Bez újmy na obecnosti nadto předpokládejme, že na počátku je v zásobníku řetěz S# (a nikoliv pouze symbol S), přičemž výskyt symbolu # na vrcholu zásobníku nám bude signalizovat, že je na zásobník prázdný.

Důkaz, že takto sestrojený levý analyzátor Z vskutku provádí syntaktickou analýzu shora dolů jazyka L(G), nalezneme např. v [2].

x+y

Příklad 3.3.32.

Vraťme se k příkladu 3.3.30. Vytvořili jsem v něm rozborovou tabulku M. Sestrojte pro M (podle důkazu věty 3.3.31) deterministický levý analyzátor Z (tuto konstrukci zde – pro její jednoduchost neuvádíme a ponecháme ji pro čtenáře jako cvičení).Ukažme nyní jak bude vypadat syntaktická analýza (shora dolů) slova

$$dbccdc \in L(G),$$

přičemž si opět zjednodušíme situaci tím, že v zápisu konfigurací nebudeme specifikovat stav q (neboť je jediný):

$$\begin{aligned}(dbccdc, S\#, e) &\vdash (bccdc, SA\#, 1)\\ &\vdash (ccdc, AcA\#, 12)\\ &\vdash (cdc, cA\#, 124)\\ &\vdash (dc, A\#, 124)\\ &\vdash (c, A\#, 1243)\\ &\vdash (e, \#, 12434).\end{aligned}$$

Tím je překlad dokončen Levý rozbor slova dbccdc je tudíž 12434.

Ve zbývající části tohoto odstavce naše úvahy o LL gramatikách poněkud zobecníme. V případě jednoduchých LL(1) gramatik jsem požadovali, aby pravá strana libovolného pravidla začínala terminálem. Nyní od tohoto požadavku upustíme a přejdeme ke studiu tzv. LL(1) gramatik. Začneme opět velmi jednoduchým příkladem.

x+y

Příklad 3.3.33.

Uvažujme bezkontextovou gramatiku

$$G = (\{A, B, T\},\{a, (,), +\}, P, A),$$

kde množina P obsahuje pravidla:

(1) $A \rightarrow B$
(2) $B \rightarrow + B$
(3) $B \rightarrow \varepsilon$
(4) $\rightarrow$
(5) $\rightarrow (A)$.

Nejprve si všimněme, že se nejedná o jednoduchou LL(1) gramatiku, neboť pravá strana pravidla (1) nezačíná terminálem. A přece pro tuto gramatiku můžeme sestrojit rozborovou tabulku M. Symbol T, kterým začíná pravá strana pravidla (1), tvoří totiž levou stranu pravidel (4) a (5), které obě

začínají terminálem. Rozborová tabulka M pro gramatiku G bude proto vypadat takto:

	a	(	)	+	e
A	TB,1	TB,1	-	-	-
B	-	-	e,3	+TB,2	e,3
T	a,4	(A),5	-	-	-
a	x	-	-	-	-
(	-	x	-	-	-
)	-	-	x	-	-
+	-	-	-	¨x	-
#	-	-	-	-	+

Gramatika G, kterou jsme poznali v příkladu 3.3.33 je tzv. LL(1) gramatika. Než přejdeme k její formální definici zaveďme si dvě důležité funkce **FIRST** a **FOLLOW**.

Buď

$$= (\ , \ , \ ,)$$

bezkontextová gramatika. Pro $x \in (N \cup T)^*$ definujeme:

$$FIRST(x) = \{a: x =>^* ay, a \in T, y \in (N \cup T)^* \} \cup \{e: x =>^* e \},$$

tzn. že FIRST(x) je množina všech terminálních symbolů, které lze na počátku řetězu odvodit z x.

Pro $A \in N$ definujeme:

$$FOLLOW(A) = \{c: S =>^* xAy, y =>^* cz, c \in T, x, y,z \in (N \cup T)^* \},$$

tzn. že FOLLOW(*A*) je množina všech terminálních symbolů, které mohou být ve větných formách bezprostředně za symbolem *A*. Jestliže nadto existuje větná forma, jejíž poslední symbol je *A*, pak množina FOLLOW(*A*) obsahuje rovněž e.

Uvažujme gramatiku z příkladu 3.3.33. Pak např.

FIRST(TB) = { a, (},
FOLLOW(T) = {+, e }.

Rozšiřme nyní funkci FIRST resp. FOLLOW tak, aby jejich argumenty mohly být množiny slov resp. neterminálů.

Buď = (, , ,) bezkontextová gramatika a ⊆ (∪)*, ⊆ . Pak
FIRST( ) = { : ∈ FIRST(*x*) pro nějaké *x* ∈ },

FOLLOW(W) = {A: $A \in$ FOLLOW(x) pro nějaké $A \in$ }.

Nyní jsme již připraveni definovat LL(1) gramatiku.

DEF

Definice 3.3.34.

Bezkontextová gramatika = (, , ,) se nazývá LL(1) gramatika, právě když pro každý neterminál $A \in$ platí:

Jestliže $A \to x$, $A \to \in$ a $x \neq$ pak

FIRST(x FOLLOW(A)) $\cap$ FIRST(FOLLOW(A)) = $\varnothing$

Již v příkladu 3.3.33 jsme se snažili intuitivně naznačit, že pro libovolnou LL(1) gramatiku lze vytvořit rozborovou tabulku. Nyní budeme tuto myšlenku precizovat, neboť si uvedeme přímo algoritmus vytvoření rozborové tabulky pro danou LL(1) gramatiku.

Algoritmus 3.3.35

Vytvoření rozborové tabulky pro LL(1) gramatiku.

Vstup: LL(1) gramatika = (, , ,), jež má n pravidel ($n \geq 1$) očíslovaných 1, …, n.

Výstup: Rozborová tabulka pro .

Metoda: Rozborovou tabulku budeme definovat na $\cup$ $\cup$ {#} X $\cup$ {e}.

(1) Jestliže $A \to x$ je i-té pravidlo z a $\in$ FIRST(x) – {e}, pak (A,) = x, i;
(2) Jestliže $A \to x$ je i-té pravidlo z e $\in$ FIRST(x) a $b \in$ FOLLOW(A), pak
M(A, b) = x, i;
(3) Jestliže $\in$, pak (,) = X;
(4) (#, e) = +;
(5) Ve všech zbývajících případech je na příslušném průsečíku sloupce a řádku – (tj. signalizace chyby).

Podle právě uvedeného algoritmu si snadno ověříme, že pro LL(1) gramatiku z příkladu 3.3.35 existuje rozborová tabulka , jež je uvedena v závěru tohoto příkladu. Užití algoritmu 3.3.35 si teď ilustrujeme ještě jedním zajímavým příkladem.

x+y

Příklad 3.3.36.

Uvažujme LL(1) gramatiku G = ({A, B, C, D, F}, {a, +, *, (,)}, P, A), kde P obsahuje osm pravidel:

(1) A $\to$ CB
(2) B $\to$ +CB
(3) B $\to$ e
(4) C $\to$ FD

(5) D → *FD
(6) D → e
(7) F → (A)
(8) F → a.

Všimněme si, že potřebujeme znát následující množiny symbolů, abychom byli sto podle výše uvedeného algoritmu sestrojit rozborovou tabulku M pro G.

FIRST(CB) = { a, (},
FIRST(FD) = { a, (},
FOLLOW(B) = { e,) },
FOLLOW(D) = { e,), + },
FIRST(+CB) = { + },
FIRST(*FD) = { * },
FIRST((A)) = { (},
FIRST(a) = { a }.

Teď již můžeme podle algoritmu 3.5.35 sestrojit pro G rozborovou tabulku M:

	a	+	*	(	)	e
A	CB, 1	-	-	CB, 1	-	-
B	-	+CB, 2	-	-	e,3	e,3
C	FD, 4	-	-	FD, 4	-	-
D	-	e, 6	FD, 5	-	e, 6	e, 6
F	a, 8	-	-	(A), 7	-	-
a	X	-	-	-	-	-
+	-	X	-	-	-	-
*	-	-	X	-	-	-
(	-	-	-	X	-	-
)	-	-	-	-	X	-
#	-	-	-	-	-	+

Ve výše uvedeném příkladě 3.3.36 jsme prováděli výpočet funkcí FIRST a FOLLOW, aniž bylo ovšem zcela jasné, jakým způsobem. Proto si nyní pro výpočet těchto funkcí uvedeme algoritmy.

Algoritmus 3.5.37.
Výpočet funkce FIRST.

Vstup: Bezkontextová gramatika = (, ,) a řetěz $x \in (\cup)^*$.

Výstup: FIRST(x).

Metoda:
(1) FIRST(x) = ∅

(2) Následující pravidla (i) – (iv) opakujeme (v uvedeném pořadí) tak dlouho, dokud je možné do některé z množin FIRST (viz níže) zavést e nebo nový terminál:

(i) jestliže $x \in T$, pak $FIRST(x) = \{x\}$;

(ii) jestliže $x \in N$ a $x \rightarrow e \in P$ nebo $x = e$,
pak $FIRST(x) = FIRST(x) \cup \{e\}$;

(iii) jestliže $x \in N$ a $x \rightarrow ay$, $a \in T$, $y \in (N \cup T)^*$ nebo $x = ay$,
pak $FIRST(x) = FIRST(x) \cup \{a\}$;

(iv) jestliže $X \in N$ a $X \rightarrow A_1A_2\ldots..A_k \in P$, $A_i \in N \cup T$, $1 \leq f \leq k$, $k \geq 1$
pak pro všechna n ($\leq$ k) takknn FIR5.1525(k)722]TJ /R12 8.04 Tf 0.999418 0 0 1 356

pakuj

jR)e.74( )z6.2659(y)-(

(2) Jestliže $A \to uXv \in P$, $A \in N$, u, v, $\in (N \cup T)^*$, pak do FOLLOW(X) přidáme všechny terminály z FIRST(v);
(3) Jestliže $A \to uX \in P$ nebo $A \to uXv \in P$ kde $e \in$ FIRST(v), $A \in N$, v, $u \in (N \cup T)^*$, pak do FOLLOW(X) přidej FOLLOW(A).

Opět si předvedeme, jak lze podle uvedeného algoritmu sestrojit množiny FOLLOW(B), FOLLOW(D) z přikladu 3.3.36.

x+y

Příklad 3.3.40.
Uvažujme LL(1) gramatiku F z příkladu 3.3.36 a sestrojme dle algoritmu 3.3.39 množiny FOLLOW(B) a FOLLOW(D):

FOLLOW(B) = FOLLOW(A) = { e } ∪ {) } = {e,) };

FOLLOW(D) = FOLLOW(C) = (FIRST(B) – { e }) ∪ FOLLOW(A) = ({ +, e } – { e }) ∪ { e,) } = { +,), e }.

Pro danou LL(1) gramatiku jsme vždy schopni sestrojit příslušnou rozborovou tabulku. Nyní si uveďme algoritmus, jenž na základě takovéto rozborové tabulky provádí syntaktickou analýzu. Tento algoritmus bude rozdělen do dvou základních kroků. V prvém budeme definovat (obecně nedeterministický) levý analyzátor. V druhém krku využijeme čtených symbolů vstupního řetězu k tomu, aby sestrojený analyzátor prováděl syntaktickou analýzu deterministicky, tj. aby mohl z libovolné konfigurace provést nejvýš jeden takt.

Algoritmus 3.3.41.
Syntaktická analýza shora dolů pro LL(1) gramatiky.

Vstup: Rozborová tabulka pro LL(1) gramatiku = (, , ,), která má n pravidel očíslovaných 1, …, *n*; řetěz ∈ *.

Výstup: Levý analyzátor T, který provádí syntaktickou analýzu (shora dolů) jazyka L(G), tj. definuje překlad
$P_e(T) = \{(x, v): x \in L(G), v$ je levý rozbor x $\}$;
levý rozbor řetězu, jestliže $x \in L(G)$, jinak chybová signalizace <CHYBA>.

Metoda:
(1) Definujme zásobníkový převodník
T = ({q}, T, N ∪ T ∪ {#}, {<CHYBA>} ∪ {1, …, n}, *δ*, q, S, ∅), kde *δ* definujeme:
(i) jestliže M(A, a) = x, i, $A \in N$, $a \in T$, $x \in (N \cup T)^*$, $i \in \{1, \ldots, n\}$, pak $(q, x, i) \in \delta(q, e, A)$;
(ii) jestliže M(a, a) = X, $a \in T$, pak $\delta(q, a, a) = (q, e, e)$;
(iii) jestliže M(A, a) = -, $A \in N \cup T \cup \{\#\}$, $a \in T \cup \{e\}$, pak $\delta(q, e, A) = \{(q, e, <CHYBA>)\}$.

(2) Syntaktická analýza bude provedena deterministicky, tj. levý analyzátor bude mít na základě právě čteného symbolu z jednoznačně

určenou funkci δ, pomocí které má ze stávající konfigurace provést takt (v zápisu konfigurací budeme opět pro jednoduchost vynechávat stav q, neboť je jediný).

(a) jestliže T je konfigurace (av, Ay, z), a $\in$ T, A $\in$ N, v $\in$ T^*, y$\in$ $(N \cup T)^*$ {#}, z $\in$ {1, …, n}* a M(A, a) = x, i, i $\in$ {1, …, n}, x $\in$ $(N \cup T)^*$, pak (av, Ay, z) $\vdash$ (av, xy, zi) podle (q, x, i) $\in$ δ (q, e, A);

(b) jestliže T je v konfiguraci (av, ay, z) a M(a, a) = X (pro všechny symboly platí totéž co v (a)), pak (av, sy, z) $\vdash$ (v, y, z) podle (q, e, e) $\in$ δ (q, a, a);

(c) v konfiguraci (e, #, z), z $\in$ {1, …, n}*, syntaktická analýza končí a z je levý rozbor w (tato situace odpovídá v rozborové tabulce polí M(#, e) = +);

(d) jestliže T je v konfiguraci (av, Ay,z) a M(A, a) = - (pro všechny symboly platí totéž co v (a)), pak (av, Ay, z) $\vdash$ (v, y, z <CHYBA>), což odpovídá chybové signalizaci.

Proveďme si nyní syntaktickou analýzu shora dolů podle algoritmu 3.3.41 pro nějaký konkrétní jazyk.

x+y

Příklad 3.3.42.

Vraťme se k rozborové tabulce z příkladu 3.3.36 a ukažme si jak se podle algoritmu 3.3.41 provede syntaktická analýza shora dolů (tzn. nalezení levého rozboru) věty

(+) * $\in$ ( ).

Levý analyzátor , sestrojený podle prvního kroku algoritmu 3.3.41 provede pro (+) * tuto posloupnost taktů:

((+) * , *A*#, e) $\vdash$ ((+) * , *CB*#, 1)
$\vdash$ ((+) * , *FBD*#, 14)
$\vdash$ ((+) * , (*A*)*DB*#, 147)
$\vdash$ (+) * , *A*)*DB*#, 1471)
$\vdash$ (+) * , *CB*)*DB*#, 14714)
$\vdash$ (+) * , *FDB*)*DB*#, 147148)
$\vdash$ (+) * , *DB*)*DB*#, 147148)
$\vdash$ (+) * , *DB*)*DB*#, 147148)
$\vdash$ (+) * , *B*)*DB*#, 1471486)
$\vdash$ (+) * , +*CB*)*DB*#, 14714862)
$\vdash$ ( ) * , *CB*)*DB*#, 14714862)
$\vdash$ ( ) * , *FDB*)*DB*#, 147148624)
$\vdash$ ( ) * , *DB*)*DB*#, 1471486248)
$\vdash$ ( ) * , *DB*)*DB*#, 1471486248)
$\vdash$ ( ) * , *B*)*DB*#, 1471486248)

⊢ ( ) * , *B*)*DB*#, 14714862486)
⊢ ( ) * ,)*DB*#, 147148624863)
⊢ (* , *DB*#, 147148624863)
⊢ (* , **FDB*#, 1471486248635)
⊢ (, *FDB*#, 1471486248635)
⊢ (, *DB*#, 14714862486358)
⊢ (e, *DB*#, 14714862486358)
⊢ (e, *B*#, 147148624863586)
⊢ (e, #, 1471486248635863).

Tím jsme získali pro větu (+) * její levý rozbor: 1471486248635863.
Výše uvedené algoritmy ukazují, jak jsou (jednoduché) LL(1) gramatiky pro praxi důležité. Nicméně, tyto gramatiky jsou vlastně speciálním případem tzv. LL gramatik, které si teď budeme definovat. Nejprve si ale rozšíříme definici FIRST a FOLLOW.

DEF

Definice 3.3.43.

Buď = (, , ,) bezkontextová gramatika, $x \in (\cup)^*$, $A \in$. Pak definujeme funkce FIRST a FOLLOW (kde je celé číslo) takto:

FIRST (x) = { : $\in$ *, | | = , $x \Rightarrow^*$ pro nějaké $\in (\cup)^*$} ∪ { : $\in$ *, | | < , $x \Rightarrow^*$ };
FOLLOW (A) = { : $\in$ *, $\Rightarrow^*$ A , | | = , , $\in (\cup)^*$} ∪ { : $\in$ *, $\Rightarrow^*$ A , | | < , $\in (\cup)^*$}.

DEF

Definice 3.5.44.

Buď ≥ 0 nějaké pevné celé číslo, = (, , ,) bezkontextová gramatika a nechť pro libovolné dvě levé derivace tvaru:

(i) $\Rightarrow^* A \Rightarrow \Rightarrow^* x$,
(ii) $\Rightarrow^* A \Rightarrow \Rightarrow^* x$,
x, x, , $\in$ *, , , $\in (\cup)^*$, $A \in$, pro které FIRST (x) = FIRST (x) platí: = . Pak nazýváme LL( ) gramatika.

Bezkontextová gramatika je LL gramatika, jestliže je LL( ) gramatika pro nějaké ≥ 0.
Jinak řečeno: Mějme bezkontextovou gramatiku = (, , N), řetěz A , $\in$ *, $A \in$, $\in (\cup)^*$ a prefix slova odvozeného z A , který je tvořen zřetězením terminálů, tj. $\in$ *, | | = (pokud ovšem takovýto prefix vůbec existuje). Existuje-li nejvýše jedno pravidlo, které lze použít pro předpis A tak, aby byl odvozen řetěz terminálů s prefixem a jestliže tímto způsobem musí být provedena libovolná derivace slova nad , začínajícího prefixem , pak se nazývá LL( ) gramatika.

Metody syntaktické analýzy, které jsme si v tomto odstavci uvedli, lze zobecnit i pro LL( ) gramatiky. Takto rozšířené algoritmy syntaktické analýzy shora dolů čtenář nalezne např. v [2] a [17].

3.3.9. Syntaktická analýza zdola nahoru

Podobně jako jsme v případě syntaktické analýzy shora dolů hledali prostředky, které věty daného bez kontextového jazyka překládají na jejich levé rozbory, budeme nyní zase zkoumat prostředky, které překládají věty bez kontextového jazyka na jejich pravé rozbory, neboť právě to je úkolem syntaktického analyzátoru pracujícího zdola nahoru. Prvým takovýmto prostředkem bude jednoduché syntaxí řízené překladové schéma.

Věta 3.3.45.
Buď bezkontextová gramatika. Pak existuje jednoduché syntaxí řízené překladové schéma tak, že

$$) = \{(x,): x \in ( ) \text{ a } \text{ je pravý rozbor } x\}.$$

Důkaz. Buď

$$= (, ,)$$

bezkontextová gramatika s n pravidly očíslovanými 1, ..., n
(pro nějaké $n \geq 1$).

$$= (, , \{1, ..., n\}, ,),$$

kde pravidla z definujeme:

jestliže $A \to x \in$ je i-té pravidlo, $x \in (\cup)^*$, $i \in \{1, ..., n\}$,
pak $A \to \overline{x}$, $\overline{x}$ i $\in$, kde $\overline{x}$ je slovo x, ze kterého jsou eliminovány všechny terminály.

Důkaz, že

$$) = \{(x,): x \in ( ), \text{ je pravý rozbor } x\}$$

je ponechán jako cvičení (lze provést snadno pomocí indukce).

Všimněme si, že důkaz věty 3.3.45, je podobný jako důkaz věty 3.3.22. Pouze na pravé straně pravidel se nyní nepíší jejich čísla (pravidel) zleva, nýbrž zprava.
Konstrukci uvedenou v důkazu věty 3.3.45 si nyní ilustrujme příkladem.

x+y

Příklad 3.3.46.
Uvažujme gramatiku z příkladu 3.3.23 a větu

$+ \quad * \quad \in \quad (\quad);$

Její pravá derivace je (jako obvykle budeme vpravo od jednotlivých kroků derivace psát čísla aplikovaných pravidel):

$$\begin{aligned} E &\Rightarrow E + B && (1)\\ &\Rightarrow E + B * F && (3)\\ &\Rightarrow E + B * && (6)\\ &\Rightarrow E + F * && (4)\\ &\Rightarrow E + \quad * && (6)\\ &\Rightarrow B + \quad * && (2)\\ &\Rightarrow F + \quad * && (4)\\ &\Rightarrow \quad + \quad * && (6) \end{aligned}$$

Pravý rozbor věty $+ \quad *$ (tzn. obrácená posloupnost pravidel užitých při její pravé derivaci) je tedy 64264631.

Sestrojíme nyní jednoduché syntaxí řízené překladové schéma : podle konstrukce uvedené v důkazu věty 3.3.45:

$$= (\{E, B, F\}, \{+,*,(,), \quad\}, \{1, ..., 6\}, \clubsuit, E\}$$

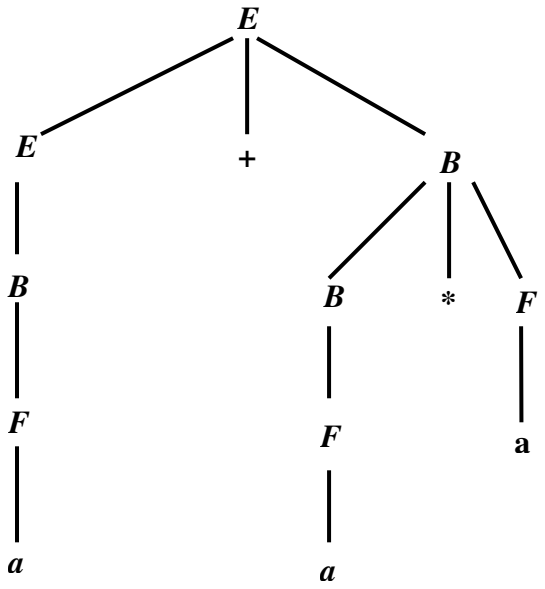
kde $\clubsuit$ obsahuje pravidla:

$E \rightarrow E + B, EB1$
$E \rightarrow B, B2$
$E \rightarrow B*F, BF3$
$B \rightarrow F, F4$
$E \rightarrow (E), E5$
$F \rightarrow \quad , 6$

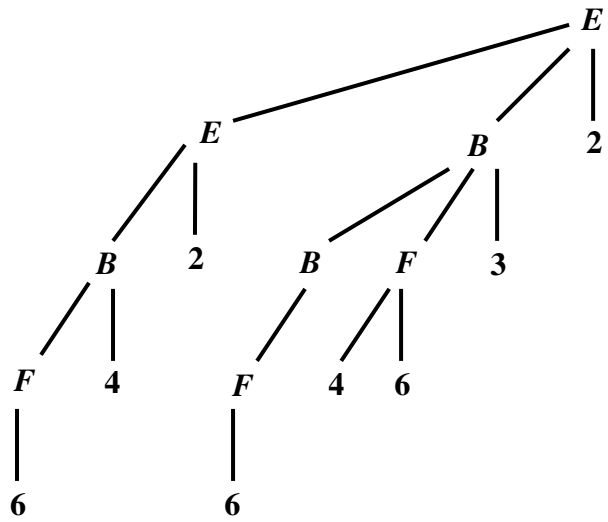
Nyní si pomocí přeložme větu $+ \quad *$ na její pravý rozbor:

$$\begin{aligned} (E, E) &\Rightarrow (E + B, EB1)\\ &\Rightarrow (E + B*F, EBF31)\\ &\Rightarrow (E + B* \quad , EB631)\\ &\Rightarrow (E + F* \quad , EF4631)\\ &\Rightarrow (E + \quad * \quad , E64631)\\ &\Rightarrow (B + \quad * \quad , B264631)\\ &\Rightarrow (F + \quad * \quad , F4264631)\\ &\Rightarrow (\quad + \quad * \quad , 64264631). \end{aligned}$$

To tedy znamená, že derivačnímu stromu ve vstupní gramatice (schématu):



odpovídá ve výstupní gramatice derivační strom:



V souvislosti se syntaktickou analýzou zdola nahoru nás ovšem v prvé řadě bude zajímat konstrukce rozšířeného zásobníkového převodníku (který překládá věty bezkontextového jazyka na jim odpovídající pravé rozbory), neboť jej můžeme chápat jako model syntaktického analyzátoru.

Věta 3.3.47.

Buď bezkontextová gramatika. Pak existuje rozšířený zásobníkový převodník tak, že

( ) = {(,): ∈ ( ) a je pravý rozbor }.

Důkaz.
Buď

$$= (\quad , \quad , \quad)$$

Bezkontextová gramatika s n pravidly očíslovanými l, ..., *n* (pro nějaké $n \geq 1$). Sestrojme rozšířený (nedeterministický) zásobníkový převodník T takto:

$$T=(\{q\}, T, N \cup T \cup \{\$\}, \{1, ..., n\}, \delta, q, \$, \Phi)$$

přičemž vrchol zásobníku je vpravo a δ definujeme:

(I) jestliže $A \to x \in P$ je *i*-té pravidlo, $i \in \{1, ..., n\}$,
pak $(q, A, i) \in \delta (q, e, x)$;
(2) jestliže $a \in T$, pak $\delta (q, a, e) = \{(q, a, e)\}$;
(3) $(q, e, \$S) = \{(q, e, e)\}$.

Důkaz, že vskutku

$$(\quad) = \{(x, \quad): x \in \quad (\quad), \quad \text{je pravý rozbor } x\}$$

je ponechán za cvičení (a nalezneme jej např. v [2]).

DEF

Definice 3.3.48.
Rozšířený zásobníkový převodník sestrojený v důkazu předcházející věty 3.3.47 se nazývá pravý analyzátor.
Syntaktickou analýzu zdola nahoru pomocí pravého analyzátoru si nyní ilustrujme příkladem.

x+y

Příklad 3.3.49.
Vraťme se ke gramatice G z příkladu 3.3.23 a sestrojme podle důkazu věty 3.3.47 pravý analyzátor T tak, že

$$P_e(T) = \{(x, v): x \in L(G) \text{ a v je pravý rozbor x}\}.$$

Pravý analyzátor (pro G) je definován:

T = {{q}, {a, *, +, (,)}, {E, B, F, a, *, +, (,), $},
{1, ..., 6}, δ, q, $, Φ},
δ (q, e, E + B)= {(q, E, 1)},
δ (q, e, B) = {(q, E, 2)},
δ (q, e, B*F) = {(q, B, 3)},
δ (q, e, F) = {(q, B, 4)},
δ (q, e, (E)) = {(q, F, 5)},
δ (q, e, a) = {(q, F, 6)},
δ (q, b, e) = {(q, b, e): b ∈ {a, *, +, (,)},
δ (q, e, $E) = {(q, e, e)}.

Podobně jako v přikladu 3.3.46 uvažujme i nyní větu a + a*a ∈ L(G) a podívejme se, jak ji pravý analyzátor T přeloží na její pravý rozbor:

(q, a + a*a, $, e) |- (q, +a*a, $a, e)

|- (q, +a*a, $F, 6)
|— (q, +a*a, $B, 64)
|— (q, +a*a, $E, 642)
|- (q, a*a, $E+, 642)
|- (q, *a, $E*a, 642)
|- (q, *a, $E+F, 6426)
|- (q, *a, $E+B, 64264)
|— (q, a, $E+B*, 64264)
|— (q, e, $E+B*a, 64264)
|— (q, e, $E+B*F, 642646)
|— (q, e, $E+B, 6426463)
|— (q, e, $E, 64264631)
|— (q, e, e, 64264631).

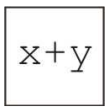
Takovýmto způsobem tedy pravý analyzátor T určí pravý rozbor 64264631 věty a + a * a.

Nyní přejděme ke zkoumání možnosti konstrukce deterministického syntaktického analyzátoru pracujícího metodou zdola nahoru.

3.3.10. LR gramatiky a deterministická syntaktická analýza zdola nahoru

Pravý analyzátor sestrojený v důkazu věty 3.3.47 je obecně nedeterministický. Pro některé speciální podtřídy bezkontextových. gramatik lze ovšem tento nedeterminismus odstranit.

Tyto gramatiky se nazývají LR gramatiky.
V tomto odstavci se budeme zabývat algoritmy, které pro LR gramatiky umožňují sestrojit příslušné deterministické pravé analyzátory, tj. modely syntaktických analyzátorů pracující zdola nahoru. Podobně jako v odstavci 3.3.9, začneme i zde jednoduchým příkladem.



Příklad 3.3.50.
Uvažujme bezkontextovou gramatika

G = ({S, A, B}, {[,], a, b, c, d} , P, S),

kde P obsahuje pravidla:

(1) S → Ab
(2) A → [B]
(3) A → a
(4) B → BAc

(5) $B \rightarrow Ad$

Sestrojme nyní podle důkazu věty 3.3.47 pravý analyzátor (tzn. rozšířený zásobníkový převodník) pro tuto gramatiko:

$T = (\{q\}, \{[,], a, b, c, d\}, \{[,], a, S, A, B, \#, b, c, d,\}, \{1, ..., 5\}, \delta, q, \#, \Phi\}$,

(i) $\delta (q, x, e) = \{(q, x, e): x \in \{[,], a, b, c, d\}$;

(ii) $\delta (q, e, Ab) = \{(q, B, 1)\}$,
$\delta (q, e, [B]) = \{(q, A, 2)\}$,
$\delta (q, e, a) = \{(q, A, 3)\}$,
$\delta (q, e, BAc) = \{(q, B, 4)\}$,
$\delta (q, e, Ad) = \{(q, B, 5)\}$;

(iii) $\delta (q, e, \#S) = \{(q, e, e)\}$.

Připomeňme znovu, že vrchol zásobníku je vpravo.
Všimněme si, že analyzátor provádí tři druhy akcí:

(1) přesun, tj. čtený symbol se přesune na vrchol zásobníku;
(2) redakce, tj. slovo, které se vyskytuje na vrcholu zásobníku a tvoří pravou stranu i-tého pravidlo se nahradí levou stranou i-tého pravidla a na výstupu se zapíše číslo pravidla, tj. i;
(3) přijatí, tj. vyprázdnění zásobníku, jedná se o poslední takt T.

Všimněme si, že náš analyzátor T je deterministický.
Výše uvedený zápis funkce δ je poměrně nepřehledný. Uveďme si proto jiné znázornění δ. Bude se jednat o tzv. tabulku akcí a tabulku přechodů, V tabulce akcí jsou uvedeny akce, které má převodník provést a v tabulce přechodů jsou určeny symboly, které mají být vloženy na vrchol zásobníku.
Nejprve si ale poněkud upravme gramatiku G. Pomocí dolních indexů si budeme specifikovat výskyt těch symbolů na pravých stranách pravidel, které se tam objevují více než jedenkrát, tj.

(1) $S \rightarrow A_1b$
(2) $A \rightarrow [B_1]$
(3) $A \rightarrow a$
(4) $B \rightarrow B_2A_2C$
(5) $B \rightarrow A_3d$.

V takto upravené gramatice máme tedy jednoznačně určen nejen symbol, ale i jeho výskyt na pravé straně.
Nyní si již pro gramatiku G můžeme uvést tabulku akcí:

Vrchol zásobníku	Prováděná akce
a	redukce (3)
b	redukce (1)
c	redukce (4)
d	redukce (5)
]	redukce (2)
[	přesun
S	přijetí
A_1	přesun
B_1	přesun
#	přesun
B_2	přesun
A_2	přesun
A_3	přesun

Jakým způsobem jsme tuto tabulku sestrojili? Především –
- protože se symbol S vyskytuje pouze na levé straně pravidla (l) a do zásobníku se tudíž vloží pouze redukcí tohoto pravidla, znamená výskyt S na vrcholu zásobníku (úspěšný) konec syntaktické analýzy (zdola nahoru), tj. akci přijetí. Dále si všimněme, že pravá strana libovolného pravidla z P končí symbolem, jenž se jinde v gramatice nevyskytuje. Této skutečnosti jsme využili k jednoznačnému určení pravidla, pomocí kterého se má provádět redukce v moment, kdy se příslušný symbol (tj. nejpravější symbol na pravé straně daného pravidla) objeví na vrcholu zásobníku. Zbývající akce (uvedené v tabulce) jsou přesuny.
U akce přesun se ještě chvíli zastavíme. Které symboly se na vrchol zásobníku ukládají? Je to buď právě čtený symbol (ze vstupní abecedy) a nebo neterminál, který tvoří levou stranu pravidla užitého při redukci. Nyní si představme, že pomocí analyzátoru překládáme slovo, které do daného bezkontextového jazyka nepatří. V takovémto případě se v zásobníku mohou objevit i řetězy, které nejsou větnou formou bezkontextové gramatiky generující náš jazyk, tzn. řetězy, které z počátečního symbolu této gramatiky vůbec nelze odvodit. Z praktického hlediska je ovšem krajně nepříjemné, že tuto skutečnost můžeme signalizovat jako chybu mnohem pozdě ji, až teprve při některé redukci, kdy zjistíme, že slovo na vrcholu zásobníku netvoří pravou stranu pravidla, podle kterého má být redukce provedena. Jistě nás ale napadne, jak lze této nepříjemné situaci poměrně snadno předejít. Stačí totiž, když budeme nějakým způsobem kontrolovat a do zásobníku ukládat pouze takové kombinace symbolů, které se vyskytují na pravých stranách pravidel. Za tímto účelem si sestrojíme (již výše zmíněnou) tabulku přechodů, pomocí které zjistíme chybu mnohem dříve.

Pro naši bezkontextovou gramatiku bude mít tabulka přechodů tvar:

	a	B	c	d	]	[	S	A	B
a								A_1	
b									
c									
d									
]									
[	a								B_1
S									
A_1		b							
A_2			c						
A_3				d					
B_1					]				
B_2	a					[		A_2	
#	a					[		A_1	

Každý řádek tabulky přechodů označuje jistý symbol, jenž se může vyskytnout jako vrchol zásobníku; sloupce jsou zase označeny symboly, které mají být - při přesunu či redukci - do zásobníku uloženy. Je-li průsečík příslušného řádku a sloupce prázdný, signalizuje to chybu.

Nyní si již uvedeme přímo algoritmus pro konstrukci tabulky akcí a přechodů, jak jsme jej intuitivně naznačili v předcházejícím příkladu 3.3.40. Nejprve si ale definujme tzv. triviální LR(0) gramatiky, pro které se tento algoritmus používá.

DEF

Definice 3.3.51.

Bezkontextová gramatika = (, , ,) se nazývá triviální LR(0) gramatika, jestliže platí:

(1) se nevyskytuje na pravé straně žádného pravidla;

(2) Každý symbol z (∪) - {} se vyskytuje na pravých stranách pravidel právě jednou.

Všimněme si, že gramatika z příkladu 3.3.50 není triviální LR(0) gramatika (neboť obsahuje symbol *A* resp. *B*, který se vyskytuje na pravých stranách pravidel třikrát resp. dvakrát) a přece jsme pro ni tabulku akcí a přechodů mohli sestrojit. Způsob, jakým jsme toho dosáhli jsme jednak již naznačili v příkladě (očíslování vícenásobného výskytu téhož symbolu na

pravých stranách pravidel) a jednak bude detailně objasněn níže (v bodě II. za příkladem 3.3.54).

Algoritmus 3.3.52.

Vytvoření tabulky akcí f a tabulky přechodů g pro triviální LR(0) gramatiky.

Vstup: Triviální LR(0) gramatika = (, , ,) s n pravidly očíslovanými 1, ..., n.

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

Metoda:

(1) Konstrukce tabulky akcí f (řádky této tabulky jsou označeny symboly z $N \cup T \cup \{\#\}$):

(a) jestliže (f) $A \rightarrow xX \in P$, $X \in N \cup T$, $i \in \{1, ..., n\}$,
pak f(X) = redukce (i);
(b) f(S) = přijetí;
(c) ve všech zbývajících případech: f(X) = přesun.

(2) Konstrukce tabulky g
(řádky jsou označeny symboly z $N \cup T \cup (\#\}$, sloupce symboly z $N \cup T$):

(a) jestliže $S \Rightarrow^* Xx$ v G, $X \in N \cup T$,
pak g(#,X) = X;
(b) jestliže $A \rightarrow xXXy$, $X, Y \in N \cup T$,
pak g(X, Y) = Y;
(c) jestliže $A \rightarrow xXBy$, $B \Rightarrow +Yz$, $B \in N$, $X, Y \in N \cup T$,
pak g(X, Y) = Y;
(d) ve všech ostatních případech: g(X, Y) = CHYBA.

Tabulek f a g sestrojených v právě uvedeném algoritmu nyní využijeme při syntaktické analýze (zdola nahoru) pro triviální LR(0) gramatiky. Později poznáme, že tento algoritmus je použitelný i pro LR(0) gramatiky. Ještě než algoritmus uvedeme, připomeňme, že tabulka akcí f a tabulka přechodů g je vlastně jiný - podstatně přehlednější - zápis funkce δ daného pravého analyzátoru T, čímž nám vlastně nahrazuje definici tohoto analyzátoru. Dále si všimněme (viz definice 3.3.48), že pravý analyzátor má jediný stav, a proto jej budeme při zápisu konfigurace pro jednoduchost vynechávat. Konfiguraci pravého analyzátoru budeme tedy rozumět trojici (z, x, v), kde z, x, v označuje po řadě: obsah zásobníku, nepřečtená část vstupního řetězu, dosud vytvořená Část výstupního řetězu. Dále se dohodněme, že (#, w, e), $w \in T^*$, resp. (#S, e, v), $v \in \{1, ..., n\}^*$, označuje počáteční resp. koncovou konfiguraci.

Algoritmus 3.3.53.

Algoritmus syntaktické analýzy pro (triviální) LR(0) gramatiky.

Vstup: Tabulka akcí f a tabulka přechodů g, pro (triviální) LR(0) gramatiku = (, ,), která má *n* pravidel očíslovaných 1, ..., *n*; vstupní slovo ∈ *.

Výstup: Jestliže ∈ ( ), pak pravý rozbor , jinak signalizace chyby.

.

Metoda:

Začínáme z počáteční konfigurace. Opakujeme (níže uvedené) body (1) a (2) tak dlouho, dokud nenastane přijetí nebo signalizace chyby. Symbol X v bodech (1), (2) označuje symbol vyskytující se na vrcholu zásobníku.

(1) (a) Jestliže f(X) = přesun , pak se přečte vstupní symbol a přesune se na vrchol zásobníku a přejde se na krok (2),

(b) Jestliže f(X) = redukce (i), i ∈ {l, ..., n} , kde

(i) $A \to w \in P$,

pak se vyloučí |w| symbolů z vrcholu zásobníku, k výstupnímu řetězu se zprava zřetězí číslo i a přejde se na krok (2),

(c) jestliže f(x) = přijetí a vstupní řetěz Je prázdný resp. neprázdný, pak se syntaktická analýza ukončí a výstupní řetěz je pravý rozbor vstupní věty resp. signalizuje se chyba;

(2) Jestliže Y je symbol, jenž má být uložen na vrchol zásobníku (tj. přečtený symbol v kroku (l.a) nebo levá strana pravidla užitého při redukci v kroku (l.b), pak:

(a) jestliže g(X,Y) = Z, pak Z uložíme na vrchol zásobníku a přejdeme na (l),

(b) jestliže položka g(X,Y) je prázdná (tj. chyba), pak signalizujeme chybu.

Uveďme si nyní příklad, jenž nám bude použití algoritmů 3.3.52, 3.3.53, ilustrovat:

x+y

Příklad 3.3.54.

Uvažujme triviální LR(0) gramatiku

G = ({S, A, B}, {a, b, c, d, h}, P, S),

kde P obsahuje pravidla

(1) $S \to Ah$
(2) $A \to aBd$
(3) $A \to c$
(4) $B \to b$.

Pomocí algoritmu 3.3.52 sestrojme tabulku akcí f:

vrchol zásobníku	prováděná akce
h	redukce (1)
b	redukce (4)
c	redukce (3)
d	redukce (2)
S	přijetí
A	přesun
B	přesun
a	přesun
#	přesun

a tabulku přechodů g:

	h	a	b	c	d	S	A	B
h								
a			b					B
b								
c								
d								
S								
A	h							
B					d			
#		a		c		S	A	

Nyní si ukažme, jak pomocí algoritmu 3.3.53 nalezneme pravý rozbor věty abdh ∈ L(G):

(#, abdh, e) |— (# a, bdh, e)
|— (# ab, dh, e)
|— (# aB, dh, 4)
|— (# aBd, h, 4)
|— (# A, h, 42)
|- (# Ah, e, 42)

$\vdash$ (#S, e, 421).

Pravý rozbor věty abdh je tudíž řetěz 421.

Prakticky význam triviálních LR(0) gramatik není velký. Zásadním důvodem je fakt, že libovolná triviální LR(0) gramatika generuje pouze konečný jazyk (velmi jednoduchý důkaz této skutečnosti nalezneme např. v [14]). Každý jazyk generovaný triviální LR(0) gramatikou lze proto akceptovat vhodným (deterministickým) konečným automatem. Problém konstrukce syntaktického analyzátoru pro triviální LR(0) gramatiku lze pak ovšem podstatným způsobem zjednodušit, neboť jej lze zredukovat na problém konstrukce konečného převodníku, jak jsme jej prodiskutovali v kapitole 3.2. Jistě nás proto napadá otázka: Proč jsme se vůbec triviálními LR(0) gramatikami zabývali a uváděli si pro ně algoritmy 3.3.52 a 3.3.53? Odpověď na tuto otázku shrneme do dvou bodů:

I. Algoritmus syntaktické analýzy 3.3.53 lze použít i pro LR(0) gramatiky, které budeme definovat níže.
II. Již v přikladu 3.3.50 jsme si ukázali, že tabulku přechodů lze sestrojit i pro některé gramatiky, které nejsou triviální LR(0) gramatiky (protože se v nich některé symboly vyskytují na pravých stranách pravidel vícekrát než jednou). V takovémto případě postupujeme přesně stejně jako v příkladě 3.3.50, tj. změníme tabulku přechodů tak, že do zásobníku ukládáme nejen jednotlivé symboly gramatiky, ale i informace o jejich umístění na pravých stranách pravidel (jednoduše si vícenásobné výskyty stejných symbolů na pravých stranách pravidel očíslujeme). Pak již můžeme použít algoritmu 3.3.52 (2) ke konstrukci tabulky přechodů. Tak např. se velmi snadno přesvědčíme, Že pomocí této úpravy gramatiky G z příkladu 3.3.50 a algoritmu 3.3.52 (2) vskutku dospějeme k tabulce přechodů, jež je v závěru zmíněného příkladu uvedena.
Zcela analogicky, jako lze pomocí očíslování symbolů na pravých stranách pravidel použít algoritmus 3.3.52 (2) pro konstrukci tabulky přechodů, můžeme pomocí stejné úpravy sestrojit podle algoritmu 3.3.52 (1) i tabulku akcí - viz příklad 3.3.50.

x+y

Příklad 3.3.55.
Ukažme si nyní, jak lze pomocí algoritmu 3.3.53, tabulky akcí a tabulky přechodů (z příkladu 3.3.50) provést syntaktickou analýzu věty

[ad] b $\in$ L(G),

kde G je gramatika z příkladu 3.3.50:

(#, [ad] b, e) $\vdash$ (# [, ad] b, e)
$\vdash\!\!-$ (# [a, d] b, e)
$\vdash\!\!-$ (# [A_3, d] b, 3)
$\vdash$ (# [A_3d,] b, 3)
$\vdash$ (# [B_1,] b, 35)
$\vdash$ (# [B_1], b, 35)

$\vdash$ (# A_1, b, 352)
$\vdash$ (# A_1b, e, 352)
$\vdash$ (# S, e, 3521).

Sestrojení tabulky akcí a přechodů podle algoritmu 3.3.52 pro triviální LR(0) gramatiky je skutečně velmi jednoduché. Ovšem pro gramatiky, které nejsou triviální LR(0) gramatiky, je sestrojení těchto tabulek podle algoritmu 3.3.53 (po příslušné, výše uvedené úpravě) natolik obtížné, že je v praxi téměř nepoužitelné. Tuto skutečnost jsme si ostatně jistě ověřili již v případě tak jednoduché gramatiky, jako byla gramatika G z příkladu 3.3.50. Proto si budeme nyní presentovat jinou metodu, která konstrukci akcí a tabulky přechodů zjednoduší.
Celý postup si opět nejprve intuitivně naznačíme vhodným příkladem.

Příklad 3.3.56.
Uvažujme bezkontextovou gramatiku

$$G = (\{S, A, B\}, \{a, b, c\}, P, S),$$

kde P obsahuje pravidla:

(1) $S \rightarrow B$
(2) $B \rightarrow aBb$
(3) $B \rightarrow A$
(4) $A \rightarrow bA$
(5) $A \rightarrow c$.

035-2.16436( )-220.276(a)3.74(k)-0pym22116436(i)-2.1]TJ -94.4957k jer

Jestliže větnou formu S přepíšeme pomocí pravidla $S \rightarrow B_1$, získáme přímou derivaci

$$S \Rightarrow B_1$$

a tedy

$$g(\#, B) = B_1.$$

Z větné formy B_1 lze pomocí pravidel $B \rightarrow aB_2b_1$ a $B \rightarrow A_1$, získat derivace

$$S \Rightarrow B_1 \Rightarrow aB_2b_1,$$
$$S \Rightarrow B_1 \Rightarrow A_1$$

a tedy

$$g(\#, a) = a,$$
$$g(\#, A) = A_1.$$

Prvý symbol větné formy A_1 je opět neterminál a pomocí pravidel $A \rightarrow b_2A_2$, $A \rightarrow c$ dostaneme derivace

$$S \Rightarrow B_1 \Rightarrow A_1 \Rightarrow b_2A_2,$$
$$S \Rightarrow B_1 \Rightarrow A_1 \Rightarrow c$$

a odtud tedy

$$g(\#, b) = b_2,$$
$$g(\#, c) = c,$$

přičemž oba symboly - jak b_2 tak c - jsou již terminály.
Podívejme se nyní na seznam pravidel, která jsme použili pro přepis prvního neterminálního symbolu ve větné formě. Eliminujme si z těchto pravidel pomocné dolní indexy a před první symbol pravé strany si zapišme tečku (.). Takto vytvořený seznam pravidel, jenž má tvar:

$S \rightarrow .B$
$B \rightarrow .aBb$
$B \rightarrow .A$
$A \rightarrow .bA$
$A \rightarrow .c$

se nazývá množina LR(0) položek. Označme si ji symbolem #, což je symbol, který se vyskytuje na vrcholu zásobníku v moment, kdy lze provést takt podle některého z uvedených pravidel.
Které symboly se mohou v průběhu syntaktické analýzy objevit v zásobníku za symbolem #? Jsou to ty symboly, které se v uvedených LR(0) položkách vyskytují za tečkou.
Nyní si představme, že jsme jako první operaci přesunuli symbol a do zásobníku, což znamená, že jsme v pravidle $B \rightarrow aBb$ zpracovali první

symbol a. Teď budeme zpracovávat symbol B. Tento fakt si označíme pomocí LR(0) položky

$$B \to a.Bb,$$

tzn. že jsme tečku v položce

$$B \to .aBb$$

posunuli za symbol a.
Ptejme se dál: které symboly mohou být za symbolem a? Vraťme se ke krokům (2.b) a (2.c) v algoritmu 3.3.52:

(2.b): jestliže $A \to xXYy$, $X, Y \in N \cup T$,
pak $g(X,Y) = Y$;
(2.c): jestliže $A \to xXBy$, $B \Rightarrow^{+} Yz$, $B \in N$, $X, Y \in N \cup T$,
pak $g(X, Y) = Y$;

což znamená, že za symbolem a může být symbol B a všechny ty symboly, které se mohou vyskytnout na začátku některého slova, jež lze odvodit z B. Analogickým postupem jako výše tedy získáme množinu LR(0) položek:

$B \to a.Bb$
$B \to .aBb$
$B \to .A$
$A \to .bA$
$A \to .c,$

kterou si označíme symbolem a, jenž se objeví na vrcholu zásobníku.
Jestliže právě uvedený postup budeme aplikovat na všechny symboly vyskytující se za tečkami v LR(0) položkách, pak získáme sobor množin LR(0) položek pro gramatiku G.

V příkladu 3.3.56 jsme si tedy naznačili metodu vytváření souboru množin LR(0) položek. Popišme si nyní celý postup pomocí algoritmu.

Algoritmus 3.3.57.
Vytvoření souboru množin LR(0) položek.

Vstup: Bezkontextová gramatika = (, , ,).
Výstup: Soubor φ množin LR(0) položek pro G.

Metoda:
(1) Vytvoření počáteční množiny LR(0) položek M_O:
(a) $M_O = \{S \to .x: S \to x \in P\}$;
(b) jestliže $A \to .Bx \in M_O$, $B \in N$, $x \in (N \cup T)^*$, $B \to y \in P$,
pak $M_O = M_O \cup \{B \to .y\}$;
(c) opakujeme (b) dokud lze přidat novou položku do M_O;
(d) $\varphi = \{M_O\}$, kde M_O je počáteční množina.

(2) Předpokládejme, že jsme sestrojili množinu LR(0) položek M_i pro nějaké $i \geq 0$. Sestrojíme pro každý symbol $X \in N \cup T$ takový, že se vyskytuje v některé LR(0) položce v M_i za tečkou, další množinu LR(0) položek M_j, kde j je index o jednu větší než nejvyšší index dosud vytvořené množiny LR(0) položek.

Tuto konstrukci provedeme takto:

(a) $M_j = \{A \to xX.y: A \to x.Xy \in M_i\}$;
(b) jestliže $A \to x.By \in M_j, B \in N, B \to z \in P$,
pak $M_j = M_j \cup \{B \to .z\}$;
(c) opakujeme (b) dokud lze do M_j přidat nové položky;
(d) $\varphi = \varphi \cup \{M_j\}$.

(3) Opakujeme (2) pro všechny doposud vytvořené množiny M_j, dokud lze do φ zavést nové množiny M_j.

Poznamenejme, že krok (1.a) a (2.a) resp. (1.b) a (2.b) se nazývá vytváření základů resp. Uzávěrů množiny LR(0) položek.

x+y

Příklad 3.3.58.

Vraťme se k předcházejícímu příkladu 3.3.56 a pro gramatiku G z tohoto příkladu sestrojíme množiny LR(0) položek pomocí algoritmu 3.3.57:

$M_0(\#) = \{ \quad S \to .B,$
$\quad B \to .aBb,$
$\quad B \to .A,$
$\quad A \to .bA,$
$\quad A \to .c\}$

$M_1(B_1) = \{ \quad S \to B.\}$

$M_2(a) = \{ \quad B \to -a.Bb,$
$\quad B \to .aBb,$
$\quad B \to .A,$
$\quad A \to .bA,$
$\quad A \to .c\}$

$M_3(A_1) = \{ \quad B \to A.\}$

$M_4(b_2) = \{ \quad A \to b.A,$
$\quad A \to .bA,$
$\quad A \to .c\}$

$M_5(c) = \{ \quad A \to c.\}$

$M_6(B_2) = \{ \quad B \to aB.c\}$

$M_7(A_2) = \{ \quad A \rightarrow bA. \}$

$M_8(b_1) = \{ \quad B \rightarrow aBb. \}$

Připomeňme, že zápis M (X) znamená, že se symbol X bude v okamžiku syntaktické analýzy určeném tečkou vyskytovat na vrcholu zásobníku.
Na souboru množin LR(0) položek (pro danou gramatiku G) si nyní budeme definovat funkci ρ:

DEF

Definice 3.3.59.
Jestliže položky tvaru

$A \rightarrow x.Xy$

patří do M_i a základ množiny M_j byl vytvořen položkami tvaru

$A \rightarrow xX.y$

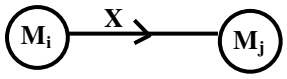
pak

$\rho(M_i, X) = M_j.$

V literatuře se nejčastěji setkáváme s reprezentací funkce ρ pomocí orientovaného, ohodnoceného (jak hranově tak uzlově) grafu. Princip takovéto reprezentace je velmi jednoduchý:
jestliže

$\rho(M_i, X) = M_j$

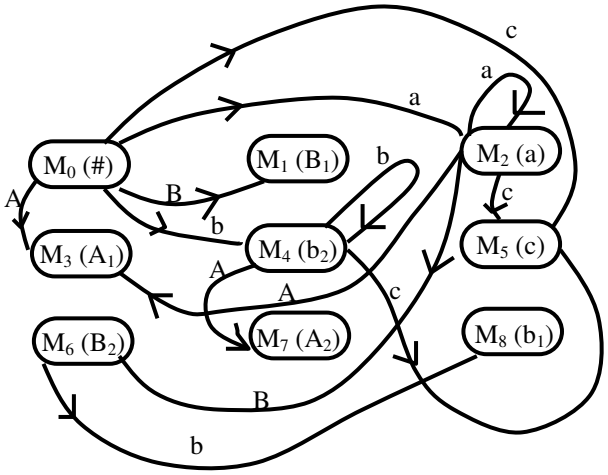
pak



Ilustrujme si to příkladem:

x+y

Příklad 3.3.60.
Uvažujme funkci ρ na souboru množin LR(0) položek x příkladu 3.3.58 a sestrojme pro ni (podle výše uvedeného postupu) graf:



Místo této grafické reprezentace je ovšem někdy výhodnější zapsat funkci ρ pomocí tabulky přechodů:

	a	b	c	B	A
$M_0(\#)$	$M_2(a)$	$M_4(b_2)$	$M_5(c)$	$M_1(B_1)$	$M_3(A_1)$
$M_1(B_1)$					
$M_2(a)$	$M_2(a)$	$M_4(b_2)$	$M_5(c)$	$M_6(B_2)$	$M_3(A_1)$
$M_3(A_1)$					
$M_4(b_2)$		$M_4(b_2)$	$M_5(c)$		$M_7(A_2)$
$M_5(c)$					
$M_6(B_2)$		$M_8(b_1)$			
$M_7(A_2)$					
$M_8(b_1)$					

Všimněme si, že se v této tabulce nevyskytuje řádek a sloupec pro počáteční symbol gramatiky. Abychom toho dosáhli, stačí místo příslušné bezkontextové gramatiky

$$G = (N, T, P, S)$$

uvažovat tzv. rozšířenou gramatiku

$$G' = (N \cup \{\check{S}\}, T, P \cup \{\check{S} \to S\}, \check{S}\},$$

kde Š je nový počáteční symbol, tj.

$$\{\check{S}\} \cap N = \varnothing.$$

V celé zbývající části této kapitoly budeme vždy automaticky předpokládat, že daná gramatika je rozšířená, přičemž pravidlo

Š → S má číslo nula, tj.

(0) Š → S.

Jestliže syntaktický analyzátor provede takt podle tohoto (tj. nultého) pravidla, budeme to považovat za úspěšné dokončení syntaktické analýzy vstupního slova, tzn. že vstupní slovo je vskutku větou dané gramatiky a na výstupu je vytvořen jeho pravý rozbor.
Pomocí souboru množin LR(0) položek si nyní budeme definovat LR(0) gramatiky.

DEF

Definice 3.3.61.

Bezkontextová gramatika = (, , ,) je LR(0) gramatika, právě když pro rozšířenou gramatiku G´ gramatiky G platí: jestliže se v některé množině souboru množin LR(0) položek (pro G') vyskytuje položka tvaru A → x. (A ∈ N ∪ {Š}, x ∈ (N ∪ T)*), pak se v této množině již nevyskytuje žádná jiná položka.
Již jsme se zmínili o tom, že pro LR(0) gramatiky, které nejsou triviální, je konstrukce tabulky akcí a přechodů podle algoritmu 3.3.52 sice možná, nicméně, zpravidla velmi obtížná. Pomocí souboru množin LR(0) položek lze však pro danou LR(0) gramatiku sestrojit tabulku akcí a přechodů mnohem snadněji, jak ukazuje následující algoritmus.

Algoritmus 3.3.62.

Konstrukce tabulky akcí f a tabulky přechodů g pro LR(0) gramatiku.

Vstup: LR(0) gramatika = (, , ,), soubor množin položek φ s počáteční množinou M_O.

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

Metoda:
1. Řádky tabulky akcí f jsou označeny stejně jako množiny položek z φ. Tabulky f sestrojíme takto:

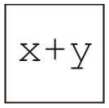
(a) jestliže A → x. ∈ M_j (pro nějakou LR(0) položku M_j)
pak f(M_j) = redukce (i), kde i je číslo pravidla A → x (i ≥ 1);

(b) jestliže Š→ S. ∈ M_j,
pak f(M_j) = přijetí;

(c) ve všech ostatních případech, f(M_j) = přesun;

2. Tabulka Přechodů g odpovídá funkci CHYBA
(a) jestliže φ (M_i, X) = M_j,
pak g(M_i, X) = M_j;
(b) jestliže φ(M_i, X) = ∅,

pak $g(M_i, X)$ bude prázdné pole, což nám bude signalizovat chybu.



Příklad 3.3.63.
Uvažujme bezkontextovou gramatiku

$$G = (\{Š, S, A, B\}, \{[,], 0, 1\}, P, Š),$$

kde P obsahuje těchto sedm pravidel:

(0) $Š \to S$
(1) $S \to A$
(2) $S \to B$
(3) $A \to [A]$
(4) $A \to 0$
(5) $B \to [B]]$
(6) $B \to 1$

Nejprve si vícenásobné výskyty symbolů na pravých stranách pravidel (0) - (6) očíslujeme:

(0) $Š \to S$
(1) $S \to A_1$
(2) $S \to B_1$
(3) $A \to [A_2]_1$
(4) $A \to 0$
(5) $B \to [B_2]_2]_3$
(6) $B \to 1.$

Pomocí algoritmu 3.3.57 dospějeme k souboru množin LR(0) položek pro G:

$M_O(\#) = \{S \to .A,$
$S \to .B,$
$A \to .[A],$
$A \to .0,$
$B \to .[B]],$
$B \to .1,$
$Š \to .S\}$

$M_1(A_1) = \{S \to A.\}$

$M_2(B_1) = \{S \to B.\}$

$M_3([) = \{ A \to [.A],$
$B \to [.B]],$
$A \to .[A],$

$A \rightarrow .0,$
$B \rightarrow .[B]],$
$B \rightarrow .1\ \}$

$M_4(0) = \{A \rightarrow 0.\}$

$M_5(1) = \{B \rightarrow 1.\}$

$M_6(A_2) = \{A \rightarrow [A.]\}$

$M_7(B_2) = \{B \rightarrow [B.]]\}$

$M_8(]_1) = \{A \rightarrow [A].\}$

$M_9(]_2) = \{B \rightarrow [B].]\}$

$M_{10}(]_3) = \{B \rightarrow [B]].\}$

$M_{11}(S) = \{\check{S} \rightarrow S.\}$

Ze tvaru těchto množin LR(0) položek vidíme, že se jedná o LR(0) gramatika. Sestrojíme nyní pomocí algoritmu 3.3.62 tabulku akcí f:

$M_O(\#)$	přesun
$M_1(A_1)$	redukce (1)
$M_2(B_1)$	redukce (2)
$M_3([)$	přesun
$M_4(0)$	redukce (4)
$M_5(1)$	redukce (6)
$M_6(A_2)$	přesun
$M_7(B_2)$	přesun
$M_8(]_1)$	redukce (3)
$M_9(]_2)$	přesun
$M_{10}(]_3)$	redukce (5)
$M_{11}(S)$	přijetí

a tabulku přechodů g:

	[	]	0	1	A	B	S
$M_O(\#)$			0	1	A_1	B_1	S
$M_1(A_1)$							
$M_2(B_1)$							

$M_3([)$	[		0	1	A_2	B_2	
$M_4(0)$							
$M_5(1)$							
$M_6(A_2)$		$]_1$					
$M_7(B_2)$		$]_2$					
$M_8(]_1)$							
$M_9(]_2)$		$]_3$					
$M_{10}(]_3)$							
$M_{11}(S)$							

Nyní se ovšem můžeme jednoduše vrátit z algoritmu 3.3.53 pro syntaktickou analýzu, o kterém již víme, že je použitelný i pro LR(0) gramatiky. Ukažme si ku příkladu, jak se pomocí tohoto algoritmu provede syntaktická analýza věty:

(#, [[0]], e) |- (# [, [0]], e)
|- (# [[, 0]], e)
|- (# [[0,]], e)
|- (# [[A_2,]], 4)
|- (# [[A_2,]$_1$,], 4)
|- (# [A_2,], 43)
|- (# [A_2]$_1$, e, 43)
|- (# A_1, e, 433)
|- (# S, e, 4331).

Vraťme se nyní k definici 3.3.61, ve které jsme zavedli LR(0) gramatiky. Pro řadu bezkontextových gramatik není podmínka, která je v této definici uvedena, splněna. Tak například se velmi často stává, že se v jedné množině LR(0) položek vyskytují položky tvaru
$A \rightarrow x.$, $B \rightarrow y.$, kde $A \neq B$, a nebo se tam vyskytují položky tvaru
$A \rightarrow x.$, $B \rightarrow y.$z. Takovéto (v praxí zcela běžné) situace nazýváme konflikty: v prvém z výše uvedených příkladů se jedná o konflikt redukce-redukce, v druhém o konflikt přesun-redukce.
V některých případech lze ovšem tyto konflikty odstranit. Jak? V zásadě lze říci, že na základě zkoumání prefixu (délky k pro nějaké $k \geq 1$) dosud nepřečtené části vstupního slova, tedy v podstatě podobně jako u LL(k) gramatik. V případě, že se nám tímto způsobem konflikty z gramatiky podaří skutečně eliminovat, budeme hovořit o tzv. jednoduchých LR(k) gramatikách, které si nyní zavedeme:

DEF

Definice 3.3.64.
Buďte = (, ,) bezkontextová gramatika,

G‘ = (N $\cup$ {Š}, T, P $\cup$ {Š $\rightarrow$ S}, Š)

rozšířená gramatika pro G, φ soubor množin LR(0) položek pro G‘ a M je libovolná množina LR(0) položek v φ. Nechť A → x.y, B → n. v jsou dvě různé položky v M. Jestliže libovolné dvě takovéto položky splňují nejméně jednu z následujících čtyř podmínek:

(1) $y \neq \varepsilon \neq v$;

(2) $y \neq \varepsilon = v$ a $FOLL0W_k(B) \cap FIRST_k(y\ FOLLCW_k(A)) = \varnothing$
pro $y \in T(N \cup T)^*$;

(3) $y = \varepsilon \neq v$ a $FOLLW_k(A) \cap FIRST_k(v\ FOLLOW_k(B)) = \varnothing$
pro $v \in T(N \cup T)^*$;

4) $y = \varepsilon = v$ a $FOLLW_k(A) \cap FOLLOW_k(B) = \varnothing$;

pak se G nazývá jednoduchá LR(k) gramatika.

Připomeňme, že funkce FIRST a FOLLOW, které jsme v definici použili, byly definovány v odstavci 3.3.9 této kapitoly.
Jednoduché LR(k) gramatiky budeme zkrácené nazývat (jak je v literatuře zvykem) SLR(k) gramatiky. Pro tyto gramatiky můžeme sestrojit tabulku akcí f a tabulku přechodů g podle následujícího algoritmu:

Algoritmus 3.3.65.
Konstrukce tabulky akcí f a tabulky přechodů g pro SLR(k) gramatiky.

Vstup: SLR(k) gramatika = (, , ,) a soubor množin LR(0) položek φ pro G.

Výstup: Tabulka akcí f a tabulka přechodů g pro G

Metoda:

(1) Konstrukce f: řádky jsou označeny stejně jako množiny položek v φ, sloupce jsou označeny řetězy nad T délky k.

(a) jestliže $A \rightarrow x.y \in M_j$, $y \in T(N \cup T)^*$, $z \in FIRST_k(y\ FOLL0W_k(A))$
pak $f(M_i, z)$ = přesun;
(b) jestliže $A \rightarrow x. \in M_i$, $A \rightarrow x$ je j-té pravidlo v P
(obsahuje-li P n pravidel, $n \geq 1$, pak
$j \in \{1, ..., n\}$); $u \in FOLLOW_k(A)$,
pak $f(M_i, u)$ = redukce (j);
(c) jestliže $Š \rightarrow S. \in M_i$,
pak $f(M_j, e)$ = přijetí;
(d) pro všechny ostatní případy: $f(M_i, *)$ = chyba.

(2) Konstrukce g odpovídá funkci φ, tj.:

(a) jestliže $\varphi(M_i, x) = M_j$, $x \in (n \cup T)$,
pak $g(M_i, x) = M_j$;
(b) jestliže $\varphi(M_i, x) = \emptyset$, $x \in (N \cup T)$,
pak $g(M_i, x)$ = chyba.

Jistě jsme si všimli, že tabulka akcí pro jednoduché LR(k) gramatiky je dvourozměrná. Z tohoto důvodu nemůžeme pro SLR(k) gramatiky použít přímo algoritmus syntaktické analýzy 3.3.53, ale musíme jej poněkud upravit. Dohodněme se přitom, že jako počáteční konfiguraci sestrojeného syntaktického analyzátoru budeme chápat trojici

(M_0, w, e),

M_0 - počáteční množina LR(0) položek, w - vstupní slovo (jež má být analyzováno), a jako koncovou konfiguraci budeme rozumět trojici

(M_i, e, v),

kde M_i je symbol na vrcholu zásobníku, při kterém došlo k při jetí a v je vytvořený výstupní řetěz (= pravý rozbor w).

Algoritmus 3.3.66.
Syntaktická analysa pro SLR(k) gramatiky.

Vstup: Tabulka akcí f, tabulka přechodů g pro SLR(k) gramatiku G = (N, T, P, Š); vstupní slovo $w \in T^*$, jehož (syntaktická) analýza (zdola nahoru) má být provedena;

Výstup: jestliže $w \in L(G)$, pak pravý rozbor w, jinak signalizace chyby.

Metoda:
Startujeme s počáteční konfigurace a opakujeme kroky (1) (2), (3) tak dlouho, dokud nenastane akce přijetí, nebo chyba. Symbol X v (níže uvedených) bodech (1), (2), (3), označuje stávající vrchol zásobníku.

(1) Určíme prefix u, $|u| = k$, z doposud nepřečtené části vstupního slova;

(2) (a) jestliže f(X, u) = přesun, pak se přečte vstupní symbol a přejde na krok (3)
(b) jestliže f(X, u) = redukce(i), (i) $A \to x \in P$,
pak ze zásobníku eliminujeme $|x|$ symbolů, k výstupnímu slovu připojíme (zprava) číslo i a přejdeme na (3);
(c) jestliže r(X, e) = přijetí a vstupní slovo w je (resp. není) přečteno do konce pak se ukončí syntaktická analýza w a výstupní slovo je pravý rozbor w (resp. signalizuje se chyba);
(d) jestliže f(X, u) = chyba, pak se signalizuje chyba.

(3) Jestliže Y je symbol, který má být uložen na vrchol zásobníku (tj. přečtený symbol, viz 2.a, nebo levá strana pravidla použitého při redukci, viz 2.b), pak:

(a) jestliže g(X, Y) = Z,
pak uložíme Z na vrchol zásobníku a přejdeme na (1);
(b) jestliže g(X, Y) = chyba, pak signalizujeme chybu (a syntaktickou analýzu ukončíme).

Výše jsme si uvedli několik algoritmů pro jednoduché LR(k) gramatiky. Ilustrujme si nyní jejich použití na příkladu:

Příklad 3.3.67.
Zvolme si záměrně gramatiku, která není LR(0) gramatikou.
Buď

$$G = \{\{Ě, E, T, F\}, \{+, *, [,], a\}, P, Ě\},$$

kde P obsahuje pravidla:

$Ě \rightarrow E$
$Ě \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow [E]$
$F \rightarrow a.$

Pomocí algoritmu 3.3.57 vytvoříme pro tuto gramatiku soubor množin LR(0) položek δ:

$M_0(\#) = \{Ě \rightarrow .E,$
$E \rightarrow .E + T,$
$E \rightarrow .T, T \rightarrow .F,$
$T \rightarrow .T * F,$
$F \rightarrow .[E],$
$F \rightarrow .a\}$

$M_1(E_1) = \{Ě \rightarrow E.,$
$E \rightarrow E. + T\}$

$M_2(T_1) = \{E \rightarrow T.,$
$T \rightarrow T. * F\}$

$M_3(F_1) = \{T \rightarrow F.\}$

$M_4(a) = \{F \rightarrow a.\}$

$M_5([) =$ $\{F \rightarrow [.E],$
$E \rightarrow .E + T,$
$E \rightarrow .T,$
$T \rightarrow .T * F,$
$T \rightarrow .F,$
$F \rightarrow .[E],$
$F \rightarrow .a\}$

$M_6(*) =$ $\{T \rightarrow T * .F,$
$F \rightarrow .[E],$
$F \rightarrow .a\}$

$M_7(E_2) =$ $\{F \rightarrow [E.],$
$E \rightarrow E. + T\}$

$M_8(T_2) =$ $\{E \rightarrow E + T.,$
$T \rightarrow T. * F\}$

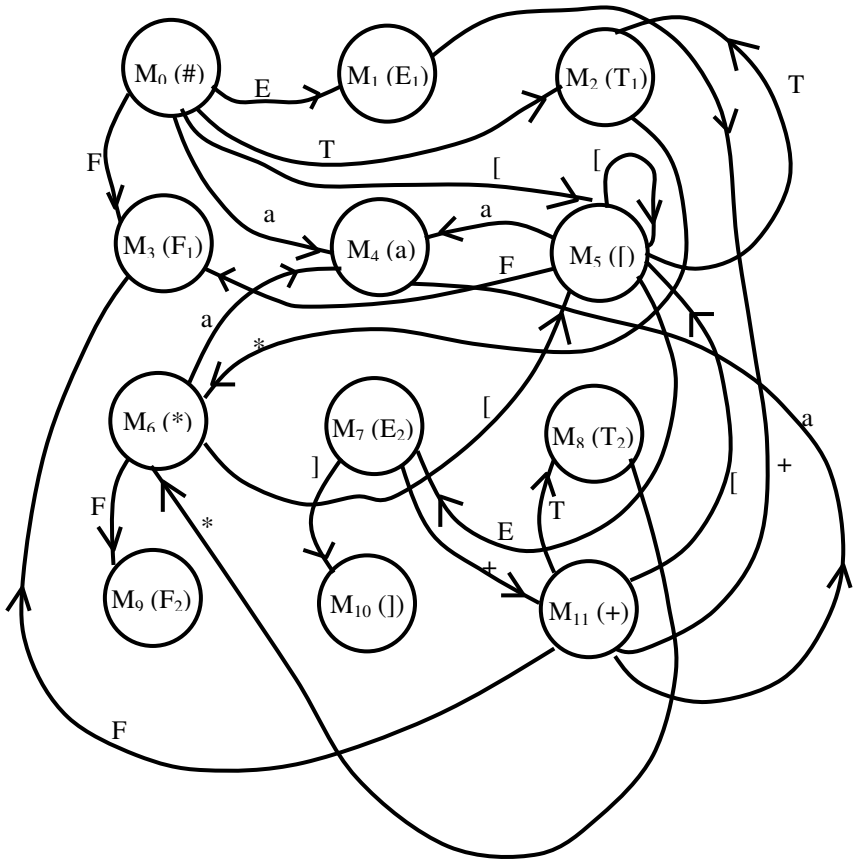
$M_9(F_2) =$ $\{T \rightarrow T * F.\}$

$M_{10}(]) =$ $\{F \rightarrow [E].\}$

$M_{11}(+) =$ $\{E \rightarrow E + .T,$
$T \rightarrow .T * F,$
$T \rightarrow .F,$
$F \rightarrow .[E],$
$F \rightarrow .a\}.$

Je zcela evidentní, že se nejedná o LR(0) gramatiku. Všimněme si ku příkladu konfliktu přesun-redukce v množině LR(0) položek $M_1(E_1)$: $\check{E} \rightarrow E.$ patří do $M_1(E_1)$ ale i $E \rightarrow E. + T \in M_1(E_1)$, tzn. nelze určit, zda máme provést redukci či přesun.

Na souboru množin LR(0) položek φ si nyní definujme funkci ρ, kterou si - jako obvykle - znázorníme pomocí orientovaného grafu. (Uzly tohoto grafu jsou označeny stejně jako množiny LR(0) položek v souboru φ. Jestliže je daná hrana ohodnocena terminálem, pak se provádí operace přesun a do zásobníku se ukládá symbol, kterým je ohodnocen uzel, do kterého hrana vstupuje. Jestliže je hrana ohodnocena neterminálem, provádíme redukci po eliminaci řetězu symbolů ze zásobníku se na jeho vrcholu objeví symbol, kterým je ohodnocen uzel, ze kterého hrana vychází a do zásobníku se uloží symbol, jímž je ohodnocen uzel, do kterého hrana vstupuje.)



Nyní pomocí algoritmu 3.3.65 sestrojíme tabulku akcí f a tabulku přechodů g:

f:	a	+	*	[	]	E
M0(#)	přesun			přesun		
M1(E1)		přesun				přijetí

M2(T1)		redukce(2)	přesun		redukce(2)	redukce(2)
M3(F1)		redukce(4)	redukce(4)		redukce(4)	redukce(4)
M4(a)		redukce(6)	redukce(6)		redukce(6)	redukce(6)
M5([)	přesun			přesun		
M6(+)	přesun			přesun		
M7(*)	přesun			přesun		
M8(E2)		přesun			přesun	
M9(T2)		redukce(l)	přesun		redukce(1)	redukce(l)
M10(F2)		redukce(3)	redukce(3)		redukce(3)	redukce(3)
M11(])		redukce(5)	redukce(5)		redukce(5)	redukce(5)

g:	E	T	F	a	+	*	[	]
M0(#)	E1	T1	Fl	a			[	
M1(E1)					+			
M2(T1)						*		
M3(F1)								
M4(a)								
M5([)	E2	Tl	F1	a			[	
M6(+)	Tl	T2	Fl	a			[	
M7(*)			F2	a			[	
M8(E2)					+			]
M9(T2)						*		
M10(F2)								
M11(])								

Nyní si již můžeme předvést syntaktickou analýzu některé věty z L(G). Uvažujeme např. slovo:

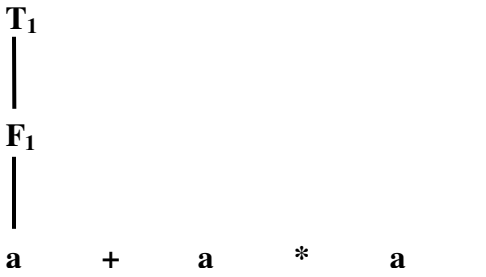
$$a + a * a \in L(G).$$

Pomocí algoritmu 3.3.66 se provede syntaktická analýza (zdola nahoru) této váty takto:

(# , a + a * a, e) |— (#a, +a * a, e)
|— (#F_1, +a * a, 6)
|— (#T_1, +a * a, 64)
|— (#E_1, +a * a, 642)
|— (#E_1+, a * a, 642)
|— (#E_1+a, *a, 642)
|— (#E_1 + F_1, *a, 6426)
|— (#E_1 + T_2, *a, 64264)
|— (#E_1 + T_2*, a, 64264)
|— (#E_1 + T_2 * a, e, 64264)
|— (#E_1 + T_2 * F2, e, 642646)
|— (#E_1 + T_2, e, 6426463)
|— (#E_1, e, 64264631).

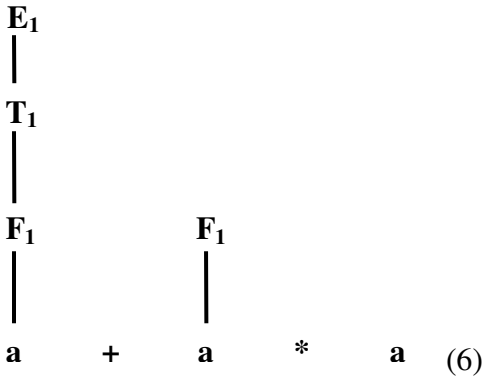
Tato posloupnost taktů odpovídá následující konstrukci (zdola nahoru) derivačního střenu pro větu a + a * a (vpravo píšeme jako obvykle číslo užitého pravidla):

(6)

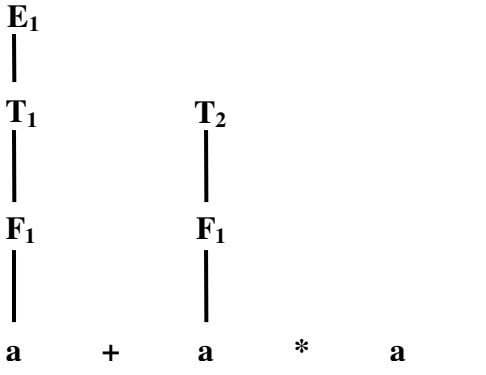


(4)

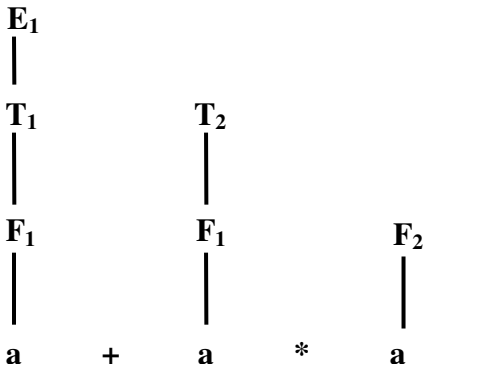
(2)


E1
T1
F1
F1
a + a * a

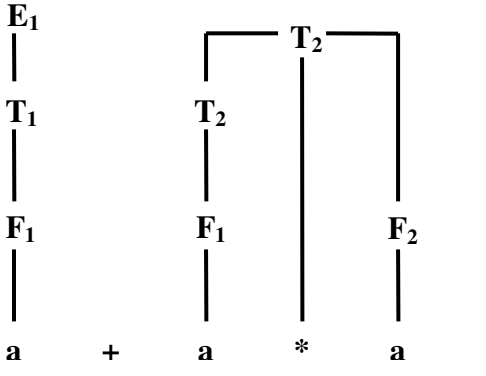
(6)


E1
T1
T2
F1
F1
a + a * a

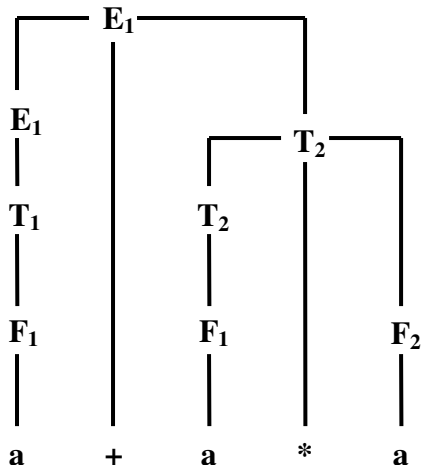
(4)


E1
T1
T2
F1
F1
F2
a + a * a

(6)


E1
T2
T1
T2
F1
F1
F2
a + a * a

(3)



(1)

Pravý rozbor věty a + a * a ∈ L(G) je tedy: 64264631, což měla syntaktická analýzy (zdola nahoru) za úkol určit.

Doposud jsme se zabývali pouze speciálními typy LR gramatik. Definujme si nyní tyto gramatiky obecně:

DEF

Definice 3.3.68.

Buď = (, , ,) bezkontextová gramatika a nechť G' = (N ∪ {Š}, T, P ∪ {Š → S}, Š) je rozšířená gramatika pro G. Gramatika G se nazývá LR(k) gramatika, pro nějaké k ≥ 0, jestliže následující tři podmínky:

(l) existuje pravá derivace $Š \Rightarrow^* xAw \Rightarrow xuw$,
(2) existuje pravá derivace: $Š \Rightarrow^* vBz \Rightarrow xuy$,
(3) $FIRST_k(w) = FIRST_k(y)$,

implikují:

(i) x = v,
(ii) A = B,
(iii) z = y.

Bezkontextová gramatika G se nazývá LR gramatika, jestliže je LR(k) gramatika pro nějaké k.

Vysvětleme si neformálně, co vlastně definice 3.3.68 říká: Jestliže xuw, xuy jsou dvě větné formy derivované pravou derivací v dané (rozšířené) gramatice, kde $FIRST_k(w) = FIRST_k(y)$ a A → u je poslední použité pravidlo v pravém odvozeni slova xuw, pak pravidlo A → u je rovněž poslední aplikované pravidlo v pravé derivaci slova xuy. Protože A může přímo derivovat u nezávisle na w, je $FIRST_k(w)$ postačující podmínka pro určení, že řetěz xu byl přímo derivován z xA.

Nyní jsme tedy uvedli zcela obecnou definici LR gramatik. I pro ně existují algoritmy syntaktické analýzy. V této práci se jimi ovšem již zabývat nebudeme, čtenář nalezne tyto algoritmy např. v [2].

3.3.11. Cvičení

1. Nalezněte jednoduchou LL(1) gramatiku pro jazyky:

(a) $\{1^n\ 0^n : n \geq 0\}$

(b) $\{\ ^R : \ \in \{0, 1\}^*\}$

(c) $\{1^n 0^n : n > 0\}$

2. Sestrojte deterministický levý analyzátor pro jazyky (a) - (c) ze cvičení 1.

3. Naprogramujte algoritmy syntaktické analýzy shora dolů (uvedené v této kapitole) v programovacím jazyku PASCAL.

4. Nalezněte LR(0) gramatiky pro jazyky:

(a) $\{1^n\ 0^n : n \geq 0\} \cup \{1^n 0^{2n} : n \geq 0\}$

(b) $\{1^n 0^n : 0 < n < n_0\}$

(c) $\{\ 1^n 0^n : n > 0\} \cup \{1^n 0^{2n} : n > 0\}$

5. Sestrojte deterministicky pravý analyzátor pro jazyky (a) - (c) ze cvičení 4.

6. Naprogramujte algoritmy syntaktické analýzy zdola nahoru (uvedené v této kapitole) v programovacím jazyku PASCAL.

3.4. Generování kódu

V této kapitole se budeme zabývat třetí a poslední fází překladu zdrojového programu – generování kódu. Jak jsme se již řekli v úvodu, tato fáze se zpravidla sama rozpadá do třech etap:

(1) generování vnitřní formy programu;

(2) optimalizace;

(3) generování cílového programu.

3.4.1. Generování vnitřní formy programu

V předcházející kapitole 3.3 jsme si řekli, že vstupem syntaktického analyzátoru je řetěz lexikálních symbolů, jenž je přeložen na řetěz reprezentující derivační strom (tj. na levý či pravý rozbor), který určuje syntaktickou strukturu daného zdrojového programu. V následující etapě (kterou si probereme v tomto odstavci) by pak měl být tento řetězec přeložen na řetěz reprezentující vnitřní formu programu. V praxi se takto ale většinou nepostupuje. Zpravidla se totiž k syntaktické analýze přímo připojí generování vnitřní formy programu, jinak řečeno, řetěz lexikálních symbolů je přímo přeložen na řetězec, který tuto vnitřní formu reprezentuje.

Ještě než si objasníme, jak se k syntaktické analýze generování vnitřní formy programu připojí, odpovězme si na tuto fundamentální otázku: proč vlastně vůbec generujeme vnitřní formu programu a ne přímo posloupnost instrukcí v cílovém jazyce (což je většinou již přímo strojový jazyk)? Existují hned dva pádné důvody, proč tomu tak je:

(1) vnitřní forma programu je nezávislá na strojovém jazyce daného počítače a tím pádem lze kompilátor velmi snadno převést do různých operačních systémů a na různé typy počítačů;

(2) ještě před vlastním vygenerováním cílového programu lze (na úrovni vnitřní formy programu) provést některé důležité optimalizace tohoto programu, jak je poznáme v následující kapitole 3.4.2.

3.4.1.1. Základní možnosti reprezentace zdrojových programů ve vnitřní formě

V tomto odstavci si uvedeme dvě nejpoužívanější vnitřní formy zdrojových programů: polský zápis a trojice.

Zabývejme se nejprve polským zápisem (který se někdy také nazývá polská notace), jenž zavedl známý polský logik Lukasiewicz.

Existují dva základní typy polského zápisu: postfixový a prefixový.

Náš výklad polského zápisu si ovšem poněkud zjednodušíme, neboť se budeme zabývat pouze zápisem aritmetických výrazů. Normální zápis aritmetických výrazů, který jsme zvyklí v matematice používat, budeme nazývat infixový.

DEF

Definice 3.4.1.

Buď B množina binárních operací (např. +, *, -, / atd.) a V množina operandů.

(1) Jestliže infixový výraz E je tvořen operandem, tj. E je prvkem V, pak jak prefixový tak postfixový polský zápis tohoto výrazu je E;
(2) Jestliže E_1bE_2 je infixový výraz, kde b je prvkem B a E_1, E_2 jsou infixové výrazy, pak:

   (a) $b\check{E}_1\check{E}_2$ je prefixový polský zápis výrazu E_1bE_2 , kde $\check{E}_1$, $\check{E}_2$ jsou prefixové polské zápisy infixových výrazů E_1 ,E_2 ;
   (b) $\hat{E}_1\hat{E}_2b$ je postfixový polský zápis výrazu $\hat{E}_1b\hat{E}_2$, kde $\hat{E}_1$ resp. $\hat{E}_2$ je postfixový polský zápis infixového výrazu E_1 resp. E_2 ;

(3) Jestliže (E) je infixový výraz, pak:
   (a) $\check{E}$ je prefixový polský zápis (E), kde $\hat{E}$ označuje prefixový polský zápis infixového výrazu E;
   (b) $\hat{E}$ je postfixový polský zápis (E), kde $\hat{E}$ označuje postfixový polský zápis infixového výrazu E.

x+y

Příklad 3.4.2.

Infixovému aritmetickému výrazu:

(a + b) * c

odpovídá prefixový polský zápis:

* + abc

a postfixový polský zápis:

ab + c * .

Postfixový polský zápis výrazů má řadu prakticky výhodných vlastností. Operandy se zde vyskytují ve stejném pořadí jako v běžném infixovém zápisu, neexistují zde závorky a nadto se operátory vyskytují vždy bezprostředně za svými operandy a to v pořadí, v jakém mají být při vyhodnocování výrazu prováděny.

Vyhodnocení výrazu v postfixovém polském zápisu by bylo jistě možné realizovat pomocí (deterministického) zásobníkového převodníku. Je to v tomto případě ovšem zbytečně silný prostředek.

Následující algoritmus, podle kterého můžeme takovéto vyhodnocení provést, bude používat jednoduše pouze zásobník Z. Postfixový polský zápis příslušného výrazu budeme psát ve tvaru posloupnosti n ($n \geq 1$) operandů a operátorů: a_1, …, a_n, přičemž a_n = # , což

je tzv. koncový symbol. I konfiguraci algoritmu budeme chápat velmi jednoduše a to jako dvojici (x, y), kde x je zbytek vstupního výrazu a y stávající obsah zásobníku Z. Na množině konfigurací pak můžeme běžným způsobem definovat relaci $\vdash$ (taktu).

Algoritmus 3.4.3.
Vyhodnocení aritmetického výrazu v postfixové polské notaci.

Vstup: Výraz v = $a_1, \ldots, a_n$ (pro nějaké $n \geq 1$) v postfixové polské notaci.

Výstup: Hodnota v.

Metoda:

(1) i:=1;
(2) čti a_i ;
(3) jestliže a_i je operand, pak:

- (i) vlož jeho hodnotu na vrchol zásobníku Z,
- (ii) i:=i+1;
- (iii) jdi na (2);

(4) jestliže a_i je operátor, pak:

- (i) vyber operandy z vrcholu Z a proveď s nimi příslušnou operaci (danou operátorem a_i),
- (ii) výsledek z (i) (což je operand) ulož na vrchol zásobníku,
- (iii) i:=i+1,
- (iv) jdi na (2);

(5) jestliže a_i = #, pak je na vrcholu zásobníku hodnota výrazu v – proveď její výstup a výpočet ukonči.

Jako obvykle si budeme použití právě uvedeného algoritmu ilustrovat příkladem. V průběhu vyhodnocování si budeme výsledek daného výrazu w na vrcholu zásobníku zapisovat vždy do hranatých závorek, tj. [w].

x+y

Příklad 3.4.4.
Uvažujme aritmetický výraz

$$a + (b + c * d).$$

Jeho postfixový polský zápis je:

$$abcd * + +,$$

což je výraz, který bude pomocí algoritmu 3.4.3 vyhodnocen takto (budeme předpokládat, že vrchol zásobníku je vpravo):

(abcd*++#, e) ⊢ (bcd*++#, a)

⊢ (cd*++#, ab)

⊢ (d*++#, abc)

⊢ (*++#, abcd)

⊢ (++#, ab [c*d])

⊢ (+#, a [b + [c*d]])

⊢ (#, [a + [b + [c*d]]]) .

Polský postfixový zápis lze rozšířit tak, aby nám umožňoval zapsat libovolný výraz či příkaz daného programovacího jazyka. Definici takto rozšířeného polského zápisu nalezneme např. v [15].

Neméně častým zápisem programu ve vnitřní formě jsou tzv. trojice, které mají obecně tvar:

(i) b, a_1, a_2;

kde (i) je pořadové číslo trojice, b je (binární) operátor, a_1, a_2 jsou operandy. Operandy instrukcí jsou buď proměnné nebo konstanty. Pokud je operandem výsledek výpočtu některé jiné trojice, pak je tento operand zapsán jako ukazatel na tuto trojici. Tyto ukazatele budeme zapisovat (podobně jako pořadová čísla trojic) v závorkách, tj. (i) nám označuje ukazatel na i-tou trojici (a nikoliv konstantu i). Uvažujme např. jednoduchý aritmetický výraz:

1 + b * c.

Pomocí trojic by tento výraz byl reprezentován:

(1) * , b, c .
(2) +, 1, (1).

Poznamenejme, že trojici (2) je prvý operand konstanta 1, zatímco druhý operand je ukazatel na první trojici.

x+y

Příklad 3.4.5.

Uvažujme přiřazovací příkaz:

A := B * (C + D).

Pomocí trojic jej zapíšeme takto:

(1) +, C, D
(2) *, B, (1)
(3) :=, A, (2).

Reprezentace vnitřní formy programu pomocí trojic má řadu výhod. Jedná se především o velmi vhodný zápis pro optimalizaci programu (jak poznáme v následujícím odstavci 3.4.2), která na etapu generování vnitřní formy programu navazuje. Za povšimnutí rovněž stojí, že tento zápis vůbec nepoužívá pomocných proměnných, čímž se snižují nároky na paměť.

Poznamenejme, že existuje celá řada dalších reprezentací vnitřní formy programu, jmenujme alespoň nepřímé trojice, stromy a čtveřice. Definice těchto zápisů (jakož i diskusí o jejich výhodách a nedostatcích) nalezneme ku příkladu v [8].

3.4.1.2. Generování vnitřní formy programu při syntaktické analýze

Co je nyní naším úkolem? Přeložit řetěz lexikálních symbolů, reprezentující zdrojový program, na řetěz reprezentující vnitřní formu tohoto programu. Již v příkladu 3.3.10 jsme si sestrojili zásobníkový převodník, jenž překládal aritmetické výrazy z infixového do postfixového zápisu. Žel, takto sestrojený (srov. důkaz věty 3.3.9) syntaktický analyzátor je v praxi téměř nepoužitelný, neboť je (v obecném případě) nedeterministický. My potřebujeme generování vnitřní formy programu připojit k deterministické syntaktické analýze, tj. určit přímo algoritmus, pomocí kterého lze daný řetěz lexikálních symbolů přeložit na řetěz zapsaný v některé vnitřní formě (např. v postfixovém polském zápisu apod.).

V tomto odstavci si uvedeme takovýto algoritmus, pomocí kterého lze připojit generování vnitřní formy programu k syntaktické analýze (shora dolů). Bude se jednat o algoritmus pro překladové gramatiky, jejichž vstupní gramatika je LL(1) gramatika. Ukážeme si, jak jej lze použít pro překlad výrazů z infixového do postfixového zápisu. Nebudeme zde ovšem pracovat s rozborovou tabulkou (pro danou LL(1) gramatiku = (, , ,), jako tomu bylo v odstavci 3.3.7), ale s tzv. redukovanou rozborovou tabulkou (která má definiční obor × (∪ {ε})). Nejprve si proto ukažme, jak pro libovolnou překladovou gramatiku (připomeňme, že překladové gramatiky a jednoduchá syntaxí řízená překladová schémata jsou ekvivalentní, tj. definují stejnou třídu překladů), jejíž vstupní gramatika je LL(1) gramatika, lze velmi jednoduše redukovanou rozborovou tabulku sestrojit.

Algoritmus 3.4.6.

Konstrukce redukované rozborové tabulky.

Vstup: Překladová gramatika = (, ∪ $_O$, Q,) taková, že její vstupní gramatika

= (, , Q ,) je LL(1) gramatika.

Výstup: Redukovaná rozborová tabulka pro .

Metoda:
Definiční obor je: $\times (V_I \cup \{\varepsilon\})$.

(1) jestliže

(i) $A \rightarrow x$ je prvkem Q,
(ii) A je prvkem $FIRST(h_I(x))$, a != e,
(iii) $x \Rightarrow^* \varepsilon$ implikuje a je prvkem FOLLOW(A),
pak M(A, a) = x;

(2) M(A, a) = CHYBA ve všech ostatních případech.

Nyní si můžeme uvést překladový algoritmus pro překladovou gramatiku G a LL(1) vstupní gramatikou.

Algoritmus 3.4.7.
Překlad pro překladovou gramatiku s LL(1) vstupní gramatiku.

Vstup: Překladová gramatika = (, $V \cup V_O$, Q, S), redukovaná rozborová tabulka pro a řetěz x je prvkem V^*, jenž má být přeložen.

Výstup: Jestliže x je prvkem ( ), pak řetěz je prvkem V_O^*, pro který platí (x,) je prvkem), jinak chybová signalizace.

Metoda:
Na začátku je v zásobníku uložen řetěz #.

(1) určíme řetěz = FIRST(x);
(2)

(a) jestliže (A,) = , pak provedeme takt: (, A , r) ⊢ (, , r);
(b) jestliže je prvkem V , pak provedeme takt:
(, , r) ⊢ (, , r);

(c) jestliže $\in V_O$

pak provedeme takt:

(, , r) ⊢ (, , r);

(d) jestliže jsme dospěli ke konfiguraci (ε, #,),
pak je překlad x, tj.

(x,) náleží)

a překlad končí;

(e) jestliže není možné provést takt podle (a), (b), (c) nebo (d),
pak překlad končí chybovou signalizací;

(3) přejdeme na (1).

Poznamenejme, že podobně jako v odstavcích 3.3.7 a 3.3.9, i ve výše uvedeném algoritmu specifikujeme konfiguraci pro jednoduchost jako trojici

$$(\quad , \quad , r)$$

kde

je dosud nepřečtená část vstupního řetězce,
je obsah zásobníku,
r je dosud vytvořená část výstupního řetězu.

Nyní si již můžeme uvést slíbený překlad, ve kterém pomocí algoritmu 3.4.7 provedeme překlad aritmetických výrazů z infixového do postfixového polského zápisu.

Příklad 3.4.8.
Mějme překladovou gramatiku

$$= (\{A, B, C, D, E\}, \{+, *, \ , (,)\} \cup \{\oplus, \otimes, @\}, Q, A),$$

kde Q obsahuje pravidla:

$A \rightarrow CB$
$B \rightarrow + C \oplus B$
$B \rightarrow \varepsilon$
$C \rightarrow ED$
$D \rightarrow * E \otimes D$
$D \rightarrow \varepsilon$
$E \rightarrow (A)$
$E \rightarrow \quad @$

Všimněme si, že vstupní gramatika

$$= (\{A, B, C, D, E\}, \{+, *, \ , (,)\}, Q , A),$$

kde Q obsahuje:

$A \rightarrow CB$
$B \rightarrow + CB$
$B \rightarrow \varepsilon$
$C \rightarrow ED$
$D \rightarrow * ED$
$D \rightarrow \varepsilon$
$E \rightarrow (A)$
$E \rightarrow$

Je vskutku LL(1) gramatika.
Podívejme se, jak v vypadá levé odvození věty

+ * ∈ ( ):

$A \Rightarrow CB$
$\Rightarrow EDB$
$\Rightarrow DB$
$\Rightarrow B$
$\Rightarrow + CB$
$\Rightarrow + ED$
$\Rightarrow + D$
$\Rightarrow + * ED$
$\Rightarrow + * D$
$\Rightarrow + *$

Protože je LL(1) gramatika, můžeme pro sestrojit podle algoritmu 3.4.6 redukovanou rozborovou tabulku:

		+	*	(	)	ε
A	*CB*			*CB*		
B		$+ C \oplus B$			ε	ε
C	*ED*			*ED*		
D		ε	$* E \otimes D$		ε	ε
E				(*E*)		

Již jsme si výše ukázali, jak vypadá odvození věty + * ve vstupní gramatice . Nyní si provedeme překlad této věty v infixovém zápisu na větu v postfixovém polském zápisu. Tento překlad se provede (podle algoritmu 3.4.7) takto:

$$
\begin{aligned}
(\ +\ \ *\ \ , A\#, \varepsilon) &\vdash (\ +\ \ *\ \ , CB\#, \varepsilon)\\
&\vdash (\ +\ \ *\ \ , EDB\#, \varepsilon)\\
&\vdash (\ +\ \ *\ \ ,\ \ @DB\#, \varepsilon)\\
&\vdash (+\ \ *\ \ , @DB\#, \varepsilon)\\
&\vdash (+\ \ *\ \ , DB\#, @)\\
&\vdash (+\ \ *\ \ , B\#, @)\\
&\vdash (+\ \ *\ \ , + C \oplus B\#, @)\\
&\vdash (\ \ *\ \ , C \oplus B\#, @)\\
&\vdash (\ \ *\ \ , ED \oplus B\#, @)\\
&\vdash (\ \ *\ \ ,\ \ @D \oplus B\#, @)\\
&\vdash (*\ \ , @D \oplus B\#, @)\\
&\vdash (*\ \ , D \oplus B\#, @@)\\
&\vdash (*\ \ , * E \otimes D \oplus B\#, @@)\\
&\vdash (\ \ , E \otimes D \oplus B\#, @@)\\
&\vdash (\ \ ,\ \ @ \otimes D \oplus B\#, @@)\\
&\vdash (\varepsilon, @ \otimes D \oplus B\#, @@)\\
&\vdash (\varepsilon, \otimes D \oplus B\#, @@@)\\
&\vdash (\varepsilon, D \oplus B\#, @@@\otimes)\\
&\vdash (\varepsilon, \oplus B\#, @@@\otimes)\\
&\vdash (\varepsilon, B\#, @@@\otimes\oplus)\\
&\vdash (\varepsilon, \#, @@@\otimes\oplus)
\end{aligned}
$$

tj.

$$(\ +\ \ *\ \ , @@@\otimes\oplus) \in \ ).$$

Generování vnitřní formy programu pochopitelně nemusí být nutně připojeno k syntaktickému analyzátoru, který pracuje metodou shora dolů. Existuje celá řada algoritmů, které umožňují připojit generování vnitřní formy zdrojového programu k syntaktické analýze zdola nahoru. Tyto algoritmy nalezneme např. v [8].

3.4.2. Optimalizace

Optimalizací programu budeme rozumět proces, ve kterém (přeuspořádáním, eliminací a nebo změnou některých operací) získáme nový, efektivnější program, jenž je ale funkčně ekvivalentní s původním.

V této kapitole si objasníme některé metody optimalizace, které se používají pro programy zapsané ve vnitřní formě (viz předcházející odstavec 3.4.1). Pro vnitřní formu programů jsou totiž optimalizační metody nejpropracovanější (vždyť je to konec konců jeden z důvodů, proč se vůbec vnitřní forma generuje), a proto také zpravidla vedou k největšímu zefektivnění zdrojového programu. Tím ovšem v žádném případě nechceme říci, že v jiné fázi překladu není optimalizace možná. Naopak – optimalizaci programu lze provádět v podstatě v libovolný moment procesu překladu: můžeme např. optimalizovat již samotný zdrojový text programu nebo jeho

syntaktickou strukturu (derivační strom) a nebo naopak až cílový program. Především by však měl autor programu sám pečlivě zvážit volbu co nejefektivnějšího algoritmu pro řešení svého problému - to za něj žádný kompilátor udělat nemůže.

V našem výkladu se pro jednoduchost omezíme opět pouze na aritmetické výrazy a přiřazovací příkazy. Jako vnitřní reprezentaci zdrojových programů si zvolíme trojice.

3.4.2.1. Předvýpočet

Zabývejme se nejprve tzv. předvýpočtem. V průběhu překladu jsou u celé řady operandů známy jejich hodnoty. Proto lze operace, v nichž se vyskytují takovéto známé operandy, provést již v průběhu kompilace a nikoliv až v době výpočtu. Objasníme si to na příkladu. Mějme posloupnost dvou přiřazovacích příkazů:

```
I := 1 + 1;
J := I - 1;
```

Kterou zapíšeme pomocí trojic:

```
(1) +, 1, 1
(2) :=, I, (1)
(3) -, I, 1
(4) :=, J, (3)
```

Především můžeme zřejmě provést již v době výpočtu trojici (1) (neboť její oba operandy jsou konstanty) a nahradit ji výslednou hodnotou. Pak ovšem můžeme provést i trojici (3), neboť známe hodnotu I (=2). Výše uvedené čtyři trojice lze tedy zredukovat na dvě:

```
(1) :=, I, 2
(2) :=, J, 1.
```

3.4.2.2. Eliminace nadbytečných operací

Další optimalizací, která se s programy ve vnitřní formě zpravidla provádí, je eliminace nadbytečných operací. Objasněme si, kterou operaci považujeme za nadbytečnou. Mějme posloupnost trojic (v pořadí, ve kterém se mají vykonávat):

```
.
.
.
(i) bi, ai1, ai2
.
.
.
(j) bj, aj1, aj2
.
.
.
```

tak, že

$$bi = bj,$$
$$ai1 = aj1,$$
$$ai2 = aj2$$

a nechť mezi i-tou a j-tou trojicí neexistuje operace, která by změnila hodnotu ai1 nebo ai2. Pak trojici (j) považujeme za nadbytečnou a z programu ji můžeme eliminovat s tím ovšem, že současně v tomto programu všechny výskyty operandu (j), tj. ukazatele na j-tou trojici, nahradíme (i), tj. ukazatelem na i-tou trojici. To znamená, že všude, kde byl jako operand uveden výsledek výpočtu trojice (j) bude nyní uveden výsledek výpočtu trojice (i).

Opět si to ilustrujme příkladem. Uvažujme posloupnost tří přiřazovacích příkazů:

```
D := D + C * B
A := D + C * B
C := D + C * B
```

Kterou zapíšeme pomocí trojic takto:

```
(1) *, C, B
(2) +, D, (1)
(3) :=, D, (2)
(4) *, C, B
(5) +, D, (4)
(6) :=, A, (5)
(7) *, C, B
(8) +, D, (7)
(9) :=, C, (8).
```

Všimněme si, že trojice (1), (4) a (7) jsou identické a nadto se hodnota proměnných C a B v celém fragmentu našeho programu, tj. mezi trojicí (1) až (9), nezmění. Trojice (4) a (7) můžeme tedy z programu eliminovat a všechny výskyty operandů (4) resp. (7) nahradit ukazatelem na trojici (1). Tím posloupnost devíti trojic zkrátíme na sedm:

```
(1) *, C, B
(2) +, D, (1)
(3) :=, D, (2)
(4) +, D, (1)
(5) :=, A, (4)
(6) +, D, (1)
(7) :=, C, (6)
```

Existují v této posloupnosti stejné trojice? Ano, jsou to (2), (4) a (6). Nyní ale musíme být opatrní. Operace určená trojicí (3) nám totiž změní hodnotu proměnné D, která je prvým operandem trojice (2). Trojici (4) tedy z programu eliminovat nemůžeme. Trojice (5) ovšem operand D nemění, a proto můžeme následující trojici (6) z programu eliminovat a druhý operand

v trojici (7) (tj.(6)) nahradit ukazatelem na trojici (4). Touto úpravou tedy konečně dospíváme k posloupnosti šesti trojic:

(1) *, C, B
(2) +, D, (1)
(3) :=, D, (2)
(4) +, D, (1)
(5) :=, A, (4)
(6) :=, C, (4)

ve které již neexistují žádné identické trojice.

3.4.2.3. Překlad bez optimalizace, jeho přednosti a nevýhody

Optimalizace (programu ve vnitřní formě) není samozřejmě etapa, která by pro překlad zdrojového programu byla naprosto nutná. Naopak, většina kompilátorů dává dokonce programátorovi možnost volby, zda optimalizace překládaného programu má - a nebo naopak nemá - být provedena. Programátor pak postupuje většinou tak, že v době ladění programu provádí překlad bez optimalizace (čímž se práce kompilátoru zrychluje), a teprve až dospěje k zcela odladěné a konečné verzi programu, zařadí do procesu překladu optimalizace. Doba překladu se tím prodlouží, ovšem výpočet podle takto zoptimalizovaného programu bude (zpravidla nezanedbatelně) efektivnější.

Podrobný výklad optimalizačních metod nalezneme např. v [8] a [20].

3.4.3. Generování cílového programu

Tato etapa je poslední v procesu překladu zdrojového programu. Zoptimalizovaný program ve vnitřní formě je zde předkládán na cílový program, což je zpravidla již přímo posloupnost strojových instrukcí popř. Posloupnost příkazů jazyka symbolických adres. Je tedy jasné, že výstup generátoru cílového programu závisí na strojovém jazyce příslušného počítače. Abychom proto mohli náš výklad dokončit, musíme si zavést svůj hypotetický počítač.

Náš počítač bude velmi jednoduchý: má paměť (s pevnou délkou slova), střadač a provádí pouze těchto šest instrukcí (P označuje obsah daného paměťového místa):

(1)	ADD	P	-	přičti k obsahu střadače P
(2)	DIVIDE	P	-	vyděl obsah střadače P
(3)	LOAD P	P	-	ulož P do střadače

(4)	MULTIPLY	P	-	vynásob obsah střadače P
(5)	STORE	P	-	ulož obsah střadače na P
(6)	SUTRACT	P	-	odečti od obsahu střadače P.

Generování cílového programu pro postfixový polský zápis proběhne prakticky stejně jako vyhodnocování tohoto výrazu (viz Algoritmus 3.4.3) až na to, že se potřebné operace neprovádí okamžitě, ale na výstupu se generují instrukce v cílovém jazyce, které pak zajistí provedení potřebných operací v době výpočtu. Takováto posloupnost instrukcí cílového jazyka tvoří cílový program.

Pro jednotlivé operátory, které budeme předkládat, tj. +, -, *, /, si zavedeme tabulky, ve kterých budou uvedeny instrukce cílového jazyka pro různé možnosti uložení operandů (viz níže). Kde to bude třeba, budeme rovněž specifikovat, zda se jedná o prvý (0_1) a nebo o druhý (0_2) operand.

V naší práci se budeme zabývat generováním cílového programu pro případ, kdy je program ve vnitřní formě

S	-	SUTRACT P (0_2) Výsledek je v S
P (0_1)	STORE Wi LOAD P (0_1) SUBTRACT Wi Výsledek je v S	LOAD P (0_1) SUTRACT P (0_2) Výsledek je v S

Tabulka pro operátor * :

*	S	P (0_2)
S	-	MULTIPLY P (0_2) Výsledek je v S
P (0_1)	MULIPLY P (0_1) Výsledek je v S	LOAD P (01) MULTIPLY P (0_2) Výsledek je v S

Tabulka pro operátor / :

/	S	P (0_2)
S	-	DIVIDE P (0_2) Výsledek je v S
P (0_1)	STORE Wi LOAD P (0_1) DIVIDE Wi Výsledek je v S	LOAD P (01) DIVIDE P (0_2) Výsledek je v S

Nyní si můžeme uvést slíbený algoritmus :

Algoritmus 3.4.9.
Generování cílového programu pro trojice

Vstup : Vnitřní forma programu zapsaná pomocí trojic.

Výstup : Cílový program

Metoda :

1) přečti trojici – pokud jsme již přečetli všechny trojice (tj. celý program ve vnitřní formě), pak je na výstupu vytvořen cílový program a generování ukončíme;

2) Podle operátora trojice vyber příslušnou tabulku. Nalezni v této tabulce položku určenou operandy trojice. Tyto operandy jsou uvedeny buď přímo v trojici a nebo – v případě, že ve zpracované trojici je použit výsledek některé předcházející trojice – na vrcholu zásobníku;

3) Jestliže je některý z operandů v zásobníku uložen ve střádači (kromě operandů, jež vstupují do právě generovaných instrukcí), ulož střádač (tj. proveď : STORE Wi; i := i + 1) a v zásobníku nahraď S označením proměnné;

4) Vygeneruj instrukce z tabulky (viz (2));

5) Zruš v zásobníku použité operandy a na vrchol zapiš adresu, kam byl uložen výsledek trojice;

6) Jdi na (I)

□

Použití algoritmu 3.4.9 je pro překlad programu zapsaného pomocí trojic na funkční ekvivalentní program ve strojovém jazyce našeho hypotetického počítače si na závěr ilustrujeme příkladem:

Příklad 3.4.10.

Mějme aritmetický výraz

(A + B) / (C – D),

Který je zapsán pomocí trojic :

(1) +, A, B
(2) -, C, D
(3) /, (1), (2)

a podívejme se, jak bude podle algoritmu 3.4.9 generování posloupnosti strojových instrukcí probíhat.

Nejprve přečteme trojici

+, A, B.

Oba dva operandy (tj. A i B) jsou v paměti, proto z tabulky pro operátor vybereme položku (v pravém dolním rohu) typu

+, P, P.

(připomeňme znovu, že P je zkratka pro obsah daného paměťového místa). Podle této položky vygenerujeme instrukce:

```
LOAD    A
ADD     B,
```

Přičemž na vrcholu zásobníku je uložen střádač (tj. S), ve kterém je výsledek těchto operací.
Nyní obsah S uložíme do pomocné proměnné W_1, tj. provedeme

STORE W_1

A v zásobníku nahradíme S označením pomocné proměnné W_1

Přečteme další, tj. druhou trojici:

-, C, D.

Vybraná položka z tabulky pro operátor – je opět typu

-, P, P

tj. oba operandy jsou uloženy v paměti. Vygenerujeme:

```
LOAD    C
SUTRACT D,
```

Obsah zásobníku je nyní

S	W_1

Přičemž ve střádači S (vrchol zásobníku) je uložen výsledek právě

Provedených operací.

Čteme poslední, třetí, trojici:

/, (1), (2).

Hodnota prvního operandu je uložena v paměti a označena pomocnou proměnnou W_1, druhý operand je ve střádači na vrcholu zásobníku.
Jinak řečeno, jedná se o operaci typu:

/, P, S.

Podle tabulky pro operátor /tedy vygenerujeme instrukce:

```
STORE    W 2
LOAD     W 1
DIVIDE   W 2.
```

Zásobník by nyní obsahoval pouze S, ve kterém je uložen výsledek.

Tedy aritmetický výraz

(A + B) / (C – D),

jenž je ve vnitřní formě reprezentován trojicemi:

```
(1)   +, A, B
(2)   -, C, D
(3)   /, (1), (2)
```

je pomocí algoritmu 3.4.9 přeložen na posloupnost strojových instrukcí:

```
LOAD     A
ADD      B
STORE    W 1
LOAD     C

SUBTRACT D
STORE    W2
LOAD     W1
DIVIDE   W2.
```

-

3.4.4. Cvičení

1. Zapište následující aritmetické výrazy v postfixovém polském zápisu :

(a) 1 + A * (6 + B / C)

(b) (1 + A) * (6 + 8 / C)
(c) (1 + (A * B + C)) / 4
(d) 1 + A * B + 8 / C

2. Proveďte vyhodnocení výrazů v postfixovém polském zápisu, které jste vytvořili v cvičení 1 (zvolte si sami hodnoty proměnných A, B, C).

3. Napište algoritmus pro generování cílového programu (ve strojovém jazyce našeho hypotetického počítače) pro postfixový polský zápis a ukažte, jak se pomocí tohoto algoritmu vytvoří cílový program pro výrazy v postfixové polské notaci, které jste vytvořili v cvičení 1.

4. Napište výrazy (a) – (d) ze cvičení 1. ve formě trojic.

5. Proveďte eliminaci nadbytečných pravidel pro programy zapsané ve formě trojic, které jsme vytvořili ve cvičení 4. (pokud lze vůbec některý z těchto programů zkrátit).

6. Proveďte generování cílového programu pro programy reprezentované trojicemi, které jsme získali ve cvičení 4.

Použitá literatura

[1] Aho, A. V., Ullman, J. D.: Principles of Compiler Design. London, Addison-Wesley, 1977.

[2] Aho, A. V., Ullman, J. D.: The Theory of Parsing, Translation and Compiling. London, Prentice Hall, 1972.

[3] Bachmann, P.: Grudlagen der Compilertechnik, ZAMM - Journal of Applied Mathematics and Mechanics 58(2). Lipsko, Tebner, 1976.

[4] Book, R. V.: Formal Language Theory, Perspectives and Open Problems. New York, Academic Press, 1980.

[5] Eilenberg, S.: Automata, Languages and Machines. Vol. A, B. New York, Academic Press, 1974.

[6] Ginsburg, S.: Algebraic and Automata: Theoretic Properties of Formal Languages. Amsterdam, Elsevier, 1975.

[7] Ginsburg, S. The Mathematical Theory of Context-Free Languages. New York, McGraw-Hill, 1966.

[8] Gries, D.: Compiler Consruction for Digital Computers. New York, Wiley, 1971.

[9] Hopcroft, J. E., Ullman, J. D.: Formal Languages and Their Ralation to Automata. London, Addison-Wesley, 1969.

[10] Hopgood, F. R. A.: Compiling Techniques. New York, MacDonald/Elsevier, 1970.

[11] Hunter, R.: The Design and Construction of Compilers. New York, Wiley, 1981.

[12] Chytil, M.: Automaty a gramatiky. Praha, SNTL, 1984.

[13] Lewis, P. M., Rosenkrantz, D. J., Stearns, R.E.: Compiler Design Theory. London, Addison-Wesley, 1976.

[14] Meduna, A.: Automata and Languages, Springer, London, 2000.

[15] Melichar, B.: Gramatiky a jazyky. Praha, Učební texty ČVUT, 1982.

[16] Melichar, B.: Překladače. Praha, Učební texty ČVUT, 1983.

[17] Raichl, J.: Překladače I. Praha, SPN, Učební texty UK, 1982.

[18] Rohl, J. S.: An Introduction to Compiler Writing. London, MacDonald, 1975.

[19] Salomaa, A: Formal Languages. New York, Academic Press, 1973.

Studijní literatura

[20] Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D.: Compilers: Principles, Techniques, and Tools (2nd Edition). London, Addison-Wesley, 2006.

[21] Hopcroft, J. E., Motwani, R., Ullman, J. D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Boston, Pearson Education, 2007.

[22] Meduna, A.: Elements of Compiler Design. New York, Taylor & Francis, 2008.

[23] Sipser, M.: Introduction to the Theory of Computation (2nd Edition). Boston, Thomson Course Technology, 2006.