SLEZSKÁ
UNIVERZITA
FILOZOFICKO-
PŘÍRODOVĚDECKÁ
FAKULTA V OPAVĚ
ÚSTAV INFORMATIKY

Šárka Vavrečková

Programování překladačů

druhá upravená verze

Opava
Poslední aktualizace: 7. listopadu 2024

Anotace: Tato skripta jsou určena studentům třetích ročníků studijních programů na Ústavu informatiky Slezské univerzity v Opavě. Pokrývají témata probíraná na přednáškách a cvičeních předmětu *Překladače*. Úkolem dokumentu je provést čtenáře obvyklou strukturou překladačů nejrůznějšího typu a naučit naučit naprogramovat si vlastní překladač.

Programování překladačů
druhá upravená verze

RNDr. Šárka Vavrečková, Ph.D.

Dostupné na: http://vavreckova.zam.slu.cz/

Ústav informatiky
Filozoficko-přírodovědecká fakulta v Opavě
Slezská univerzita v Opavě
Bezručovo nám. 13, Opava

Sázeno v systému LaTeX

Předmluva

Co najdeme v těchto skriptech

 *Rychlý náhled:* Pod pojmem překladač budeme rozumět program, který provádí transformaci dat či kódu na ekvivalentní data či kód v jiném formátu – to se týká jak překladačů běžných programovacích jazyků, tak i například příkazového řádku, webového prohlížeče,...

Cílem výuky v tomto předmětu je naučit se, jak se takový překladač navrhuje a programuje. Budeme stavět na teoretické informatice (protože struktura překladače se navrhuje právě s využitím metod vycházejících z teoretické informatiky) a postupně se dopracujeme k naprogramování vlastního překladače.

Některé oblasti jsou také „navíc“ (jsou označeny ikonami fialové barvy), ty nejsou probírány a ani se neobjeví na testech – jejich úkolem je motivovat k dalšímu samostatnému studiu nebo pomáhat v budoucnu při získávání dalších informací dle potřeby v zaměstnání.

Značení

Ve skriptech se používají následující barevné ikony:

-  *Rychlý náhled* (skript, kapitoly), ve kterém se dozvíme, o čem to bude.
-  *Klíčová slova* kapitoly.
-  *Cíle studia* pro kapitolu nám řeknou, co nového se v dané kapitole naučíme.
-  Nové *pojmy*, značení apod. jsou označeny modrým symbolem, který vidíme zde vlevo. Tuto ikonu (stejně jako následující) najdeme na začátku odstavce, ve kterém je nový pojem zaváděn.
-  Konkrétní *postupy a nástroje* (příkazy, programy, skripty). atd. jsou značeny také modrou ikonou.
-  Některé části textu jsou označeny fialovou ikonou, což znamená, že jde o *nepovinné úseky*, které nejsou probírány (většinou; studenti si je mohou podle zájmu vyžádat nebo sami prostudovat). Jejich účelem je dobrovolné rozšíření znalostí studentů o pokročilá témata, na která obvykle při výuce nezbývá moc času.

-  Žlutou ikonou jsou označeny odkazy, na kterých lze získat *další informace* o tématu. Nejčastěji u této ikony najdeme webové odkazy na stránky, kde se dané tématice jejich autoři věnují podrobněji.
-  Červená je ikona pro *upozornění* a poznámky.

Pokud je množství textu patřícího k určité ikoně větší, je celý blok ohraničen prostředím s ikonami na začátku i konci, například pro definování nového pojmu:

Definice

V takovém prostředí definujeme pojem či vysvětlujeme sice relativně známý, ale komplexní pojem s více významy či vlastnostmi.

Podobně může vypadat prostředí pro delší postup nebo delší poznámku či více odkazů na další informace. Mohou být použita také jiná prostředí:

Příklad

Takto vypadá prostředí s příkladem, obvykle nějakého postupu. Příklady jsou obvykle komentovány, aby byl jasný postup jejich řešení.

Úkol

Otázky a úkoly, náměty na vyzkoušení, které se doporučuje při procvičování učiva provádět, jsou uzavřeny v tomto prostředí. Pokud je v prostředí více úkolů, jsou číslovány.

Obsah

Úvod

Co si obvykle představíme pod pojmem překladač? Může to být překladač programovacího jazyka, ale ve skutečnosti jde o pojem mnohem širší. Překladač je vlastně program (nebo postup), který provádí transformaci dat (obecně čehokoliv). Například internetový prohlížeč provádí překlad stránky v kódu HTML do grafické podoby, které většina uživatelů rozumí lépe, databázový systém interpretuje dotazovací jazyky.

Překladač je zabudován také v každém operačním systému. Nejde jen o textové shelly, kde překladači posíláme řetězce představující příkazy, které interpretuje (například Příkazový řádek nebo BASH – Bourne-Again Shell), ale grafická nástavba je vlastně také překladač, kterému posíláme informace zadáním textu do dialogu, ťuknutím myši, přetažením objektu, ... S překladačem se setkáme i tehdy, když na Internetu zadáme k vyhledání regulární výraz nebo chceme převést obrázek ve formátu BMP na GIF.

Naším úkolem je seznámit se se strukturou překladače, jeho vlastnostmi a základními funkcemi, a dále si osvojíme základy programování jednotlivých částí překladače. Cílem je naučit se navrhnout strukturu jednoduchého, ale přesto použitelného, programovacího jazyka a pak ji přepsat na program – funkční překladač.

U studentů používajících tento studijní materiál předpokládáme znalosti v oblasti teorie formálních jazyků a automatů (předměty *Teorie jazyků a automatů I* a *II*) alespoň v rozsahu regulárních a bezkontextových jazyků (včetně odpovídajících gramatik a automatů) a schopnost programovat v alespoň jednom běžném programovacím jazyce. V textu je bez uvedení definic používáno značení, které z těchto oblastí známe. V příkladech a úkolech se dále můžeme setkat s potřebou porozumění pojmům a postupům vyučovaných v předmětu *Operační systémy I*, jejich podrobná znalost však obvykle není bezprostředně nutná.

Třebaže v oblasti překladačů je v současné době asi nejvíce používán jazyk C a jazyky s jemu podobnou syntaxí, všechny příklady v těchto skriptech jsou programovány v C a C++, místy je také použit pseudokód nebo kód v jiných programovacích jazycích. Kód příkladů je formulován tak, aby bylo snadné ho přepsat do kteréhokoliv jiného vhodného programovacího jazyka.

První kapitola uvádí čtenáře do problematiky překladačů. Je v ní načrtnuto rozdělení překladače na části, jejich funkce a vzájemná komunikace. Získáme zde obecné informace o činnosti překladače, zpracování chyb uživatele našeho programu a také o vztahu překladače k jiným programům a platformám. Krátce jsou zmíněny i editory, ve kterých uživatelé mohou psát kód následně zpracovávaný překladačem.

Druhá, třetí a čtvrtá kapitola jsou věnovány nejdůležitějším částem překladače. V každé z těchto kapitol se naučíme vytvořit návrh příslušné části překladače a podle návrhu tuto část optimálně naprogramovat. Text je doprovázen mnoha příklady a ukázkami kódu.

V páté kapitole o syntaxí řízeném překladu se naučíme všechny tři dosud probrané části překladače propojit a vytvořit kompletní funkční interpretační překladač. V příkladech najdeme většinu typických konstrukcí, které můžeme použít pro vlastní překladač.

Šestá kapitola nazvaná „Jak co naprogramovat“ obsahuje metody programování takových konstrukcí, které není nutné použít v každém překladači, přesto však existují situace, ve kterých je programátor použije – programování uživatelských datových typů, příkazů včetně větvení a cyklů, podprogramů, událostí, obecně použitelný model interpretace výrazů, generování kódu assembleru.

Následují tři přílohy se souhrnnými příklady. V každé z těchto příloh najdeme kompletní překladač od fáze návrhu až po naprogramování jeho jednotlivých částí. Překladače se navzájem liší jak svou syntaktickou strukturou, tak i způsobem návrhu a implementace. Tyto příklady slouží především jako inspirace pro vlastní překladač, který každý student v rámci cvičení vytvoří jako svou semestrální práci.

Za kapitolami (kromě příloh) vždy najdeme seznam úkolů. Tyto úkoly slouží k procvičení látky probírané v dané kapitole. Studenti prezenčního studia se s většinou z nich setkají na cvičeních, studentům kombinovaného studia doporučujeme plnit je samostatně, případně s konzultacemi s vyučujícím.

Na konci skript je seznam doporučené literatury a rejstřík. Rejstřík lze použít pro rychlé vyhledání významu a použití pojmů, seznam literatury může sloužit k prohloubení zde získaných znalostí. Definice a věty uvedené v následujících kapitolách většinou pocházejí právě z těchto zdrojů, některé byly upraveny.

Kapitola 1

Překladač a jeho struktura

Rychlý náhled: Aby bylo naprogramování překladače realizovatelné a pokud možno co nejjednodušší a nejoptimálnější, budeme chtít, aby měl určité vlastnosti. V následujícím textu se budeme zabývat návrhem vnitřní struktury překladače tak, aby program realizující tuto strukturu byl rychlý a ne moc rozsáhlý. Tato kapitola nás uvádí do problematiky, která je podrobně rozváděna v následujících kapitolách zabývajících se jednotlivými fázemi překladu.

Klíčová slova: Překladač, zdrojový program, cílový program, zdrojový jazyk, cílový jazyk, kompilátor, interpretační a hybridní překladač, lexikální, syntaktická a sémantická analýza, optimalizace, mezikód, interní kód, intermediální kód, průchod překladače, konverzační překladač, zotavení po chybě, generátor překladače, portování.

Cíle studia: Cílem této kapitoly je uvést do problematiky programování překladačů. Student se seznámí se základní strukturou překladače a rolí jednotlivých fází.

1.1 Základní pojmy

Definice (Překladač)
Překladač je program, který k libovolnému programu P_Z (zdrojový program) v jazyku J_Z (zdrojový jazyk) vytvoří program P_C (cílový program) v jazyku J_C (cílový jazyk) se stejným významem. Překladač tedy realizuje zobrazení z jazyka J_Z do jazyka J_C.

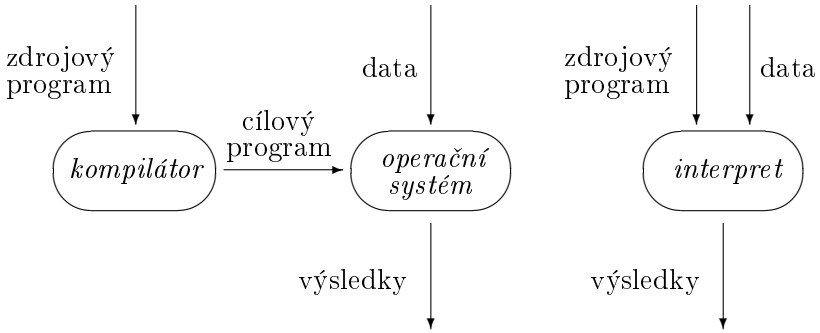
Podle typu cílového programu rozlišujeme tyto druhy překladačů:

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

interpret (interpretační překladač, někdy také interpreter) pouze interpretuje (provádí) zdrojový program pro zadaná vstupní data, tedy netvoří generovaný program, vytváří jen vnitřní reprezentaci programu pro svou vlastní potřebu (tu lze chápat jako cílový jazyk), řadíme zde shelly operačních systémů (například Příkazový řádek Windows, BASH a další shelly UNIXových systémů),

skriptovací jazyky (kromě shellů třeba Python, Ruby, PHP, Perl, Java Script), mnohé značkovací jazyky jako třeba HTML, některé čistě objektové jazyky (SmallTalk), logické programovací jazyky (Prolog), apod.,

hybridní překladače řadíme někam mezi kompilátory a interprety; generují mezikód nezávislý na operačním systému, tento mezikód je pak interpretován interpretační částí překladače instalovanou na počítači, kde mezikód spouštíme; typickými zástupci jsou Java (mezikód se zde nazývá bytecode) a C# (mezikód je typu XML).



Obrázek 1.1: Schéma kompilačního a interpretačního překladače

Na obrázku 1.1 je nákres činnosti kompilátoru a interpretačního překladače. Každý z těchto druhů je vhodný pro jinou situaci. Zatímco cílový program kompilátoru (obvykle soubor obsahující strojový kód, například EXE pro Windows) se provádí relativně velmi svižně, interpretovaný program může být pomalejší (neplatí to vždy, například interpret jazyka Perl je velmi rychlý) a pro mnoho vyšších programovacích jazyků proto nevhodný, protože překlad je prováděn při každém spuštění programu.

U kompilátorů samotný překlad probíhá jen jednou, nesouvisí se samotným prováděním programu, což umožňuje provádět i časově náročné optimalizace a kontroly (logicky překlad u interpretace nesmí trvat moc dlouho, protože je častější).

Dalšími nevýhodami interpretačního překladače jsou jeho nezbytnost při spuštění interpretovaného programu a náročnost na paměťový prostor (při běhu musí být v paměti nejen zdrojový program, ale také celý překladač). Zmíněná náročnost na paměťový prostor však není až tak velká a u současných počítačů nehraje velkou roli. Nutnost přítomnosti interpretačního překladače také nemusí být problémem u snadno dostupných překladačů (například překladače pro jazyky Perl, Python, Ruby a další jsou běžně k dispozici v linuxových distribucích).

Ale i interpretační překladač má své výhody, může například umožňovat provedení pouze malé části zdrojového programu (např. u programovacího jazyka SmallTalk), jeho vytvoření je jednodušší, programátor (autor překladače) obvykle nemusí ovládat Assembler ani strojový jazyk a při výskytu chyby můžeme spolehlivěji určit její umístění.

Navíc program uchovávaný uživatelem překladače je většinou malý textový soubor, který obvykle zabírá mnohem méně místa než obdobný cílový program přeložený kompilátorem. Zdrojový program je snadněji přenositelný (nejen proto, že se lépe „vměstná“ na jakékoliv paměťové médium, ale také je spustitelný prakticky na jakékoliv platformě – můžeme mít na strojích s různými operačními systémy nainstalován interpretační překladač pro tentýž jazyk, všechny tyto překladače přijmou tentýž zdrojový program[1]).

[1]Kompatibilita je bez problémů až na malé drobnosti, jako jsou různé způsoby označování konce řádku: ve Windows to je dvojice znaků s ASCII kódy 13 a 10 (CRLF), UNIXové systémy používají pouze 10 – LF.

V tabulce 1.1 je shrnuto srovnání prvních dvou typů překladačů, vlastnosti hybridních překladačů jsou „někde mezi“.

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

Tabulka 1.1: Srovnání vlastností kompilačního a interpretačního překladače

Některé interpretační překladače umožňují kromě interpretace také vytvoření binárního cílového kódu, například skript napsaný v Pythonu lze přeložit na spustitelný soubor pomocí nástroje v balíčku PyInstaller.

1.2 Hlavní části překladače

Překladač můžeme rozdělit na části, z nichž každá má při překladu jiný úkol. Činnost jednotlivých částí nazýváme *fáze překladu*. V překladači mohou být tyto části (obvykle zvláštní funkce) striktně odděleny nebo jsou navzájem provázány. Jsou to:

1. lexikální analyzátor,
2. syntaktický analyzátor,
3. sémantický analyzátor,
4. optimalizátor kódu,
5. generátor cílového kódu nebo interpretace.

Lexikální analyzátor má na vstupu zdrojový program celého překladače a jeho úkolem je převést ho do podoby, které rozumí ostatní části překladače. Převádí zdrojový text (případně jiné druhy dat) na posloupnost *symbolů* (atomů) – nejmenších logických částí, kterým lze přiřadit význam (například „celé číslo“, „klíčové slovo“, „levá závorka“, „relační operátor větší–rovno“, ... Každý symbol má svou identifikaci (o jaký typ symbolu jde), a pokud je to nutné, tak i atributy (sémantická data, například u symbolu „celé číslo“ přímo hodnotu tohoto čísla). Přitom odstraňuje (nebo prostě ignoruje) ty části vstupu, které nemají význam pro další překlad, například nadbytečné mezery, konce řádků, komentáře.

Syntaktická analýza je nejdůležitější částí překladu. Úkolem tohoto analyzátoru je vytvořit strukturu překládaného programu – obvykle derivační strom v některé vhodné reprezentaci. Syntaktický analyzátor skládá symboly vygenerované lexikálním analyzátorem k sobě a tvoří tak příkazy, bloky příkazů, definice proměnných či funkcí a další struktury.

Sémantický analyzátor každé skupině symbolů získané při syntaktické analýze přiřadí význam. Například při zpracování deklarace proměnné je třeba zkontrolovat, zda již není deklarována, uložit do

příslušného seznamu potřebné informace (název, typ, počáteční hodnotu, v kterém bloku je lokální, ...), může také proměnné přiřadit paměť (zatím ne přímo adresu v paměti), naopak pokud je některá proměnná použita v kódu programu, zkontroluje, zda je deklarována (u některých programovacích jazyků to není třeba), a jestli je správně použita vzhledem k jejímu deklarovanému typu, u operátoru pro sčítání zkontroluje, zda jeho operandy jsou správného typu a případně provede přetypování, ...

Výstupem sémantického analyzátoru je *intermediální kód*, což je kód již velmi podobný cílovému, má však strukturu vhodnější pro optimalizaci. Může to být zápis podobný assembleru nebo třeba dynamická struktura (dynamický seznam stromů představujících jednotlivé příkazy).

 *Optimalizátor kódu* zajišťuje, aby se používalo co nejméně pomocných proměnných pro mezivýpočty, aby se v cyklu zbytečně několikrát nevyhodnocoval tentýž výraz, jestliže hodnota jeho prvků zůstává bez změny a vyhodnocení stačí provést jednou před cyklem, apod. Použití optimalizace je charakteristické spíše pro kompilační překladače, u interpretů bývá tato fáze přeskočena.

 Program, který bez chyby prošel sémantickým analyzátorem a případně optimalizací, dále prochází *generátorem cílového programu* nebo *interpretací*. Vytváří se kód buď v jazyce symbolických instrukcí (JSI), ve vlastním jazyce sestavujícího programu (interpretační překladač) nebo přímo v jazyce stroje (strojový kód, například EXE a DLL ve Windows).

Další funkce překladače bývají zahrnuty do výše uvedených částí nebo mohou tvořit samostatnou část. Jsou to například:

- hlášení o chybách – viz kapitolu 1.5,
- informace o překladu – může být generován LOG soubor (obvykle textový soubor), který srozumitelnou formou zachycuje průběh překladu. Tento soubor je pak obvykle zobrazován editorem v samostatném okně jako „hlášení o průběhu překladu“.

 Podle závislosti na typu cílového kódu můžeme překladač rozdělit na dvě základní části:

- *Přední část* (front end) zahrnuje lexikální, syntaktickou a sémantickou analýzu. Je do značné míry nezávislá na cílovém systému, generuje vnitřní formu programu.
- *Zadní část* (back end) překladače provádí optimalizaci kódu a generuje cílový program. Tato část je již závislá na cílovém systému.

Přední část provádí analýzu („rozpitvává vstup“), zadní provádí syntézu (dává dohromady výstup).

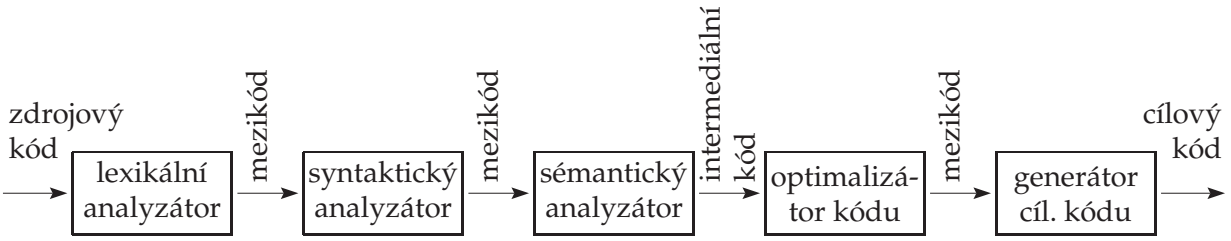
Toto rozdělení zjednodušuje vytváření překladačů téhož jazyka pro různé operační systémy. Překladače mají stejnou přední část, liší se jen v zadní části, protože typ cílového kódu bude jiný a také optimalizace mohou být určeny přímo pro danou platformu. Například pokud chceme vytvořit překladače pro tentýž programovací jazyk, které by běžely pod Windows, Linuxem i MacOS, vytvoříme jedinou přední část zahrnující lexikální, syntaktickou a sémantickou analýzu, a tři různé zadní části, které budou generovat spustitelné soubory pro tyto tři operační systémy. Námi vytvořený překladač musí samozřejmě v cílovém operačním systému také fungovat.

1.3 Průchody překladače

Překladač může pracovat tak, že nejdřív celý program projde lexikálním analyzátorem, potom je zpracován syntaktickým analyzátorem, pak opět bez přerušení dalšími fázemi překladu. Jinou možností je spolupráce těchto fází.

Definice (Průchod)

Průchodem nazýváme krok činnosti překladače, ve kterém je zpracován celý vstupní soubor kroku na výstupní soubor kroku (vstupní soubor kroku nemusí být totožný se vstupním souborem překladače, stejně tak výstupní soubor ještě nemusí být cílový kód).

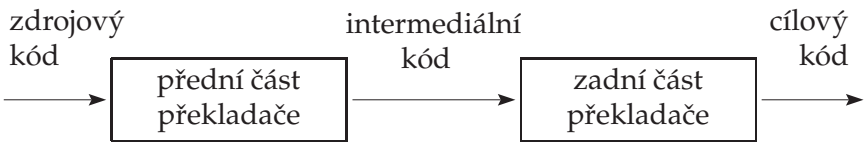


Obrázek 1.2: Víceprůchodový překladač, každá fáze v samostatném průchodu

Mezi každými dvěma průchody vznikne program určený pouze pro vnitřní potřebu, nazýváme ho *mezikód*, *interní kód* nebo *interní forma programu*, a jazyk, ve kterém je sestaven, *interní jazyk překladače*. Celý mezikód je třeba uchovávat v paměti pro zpracování v dalším průchodu. *Intermediální kód* je speciální druh mezikódu.

Průchody překladače se ne vždy kryjí s fázemi překladu. Do jednoho průchodu můžeme vtěsnat několik fází nebo jedna fáze bývá rozčleněna do více průchodů (například optimalizace). U některých jazyků je vhodné sloučit všechny fáze do jednoho průchodu, takový překladač nazýváme *jednoprůchodový*. Má jednu velkou výhodu: nevytváří mezikód, který by bylo nutné uchovávat. Jednoprůchodové překladače jsou vhodné pro jednoduché programovací jazyky, které nevyžadují důkladnou optimalizaci.

Poměrně obvyklé bývá zařazení přední části překladače (tj. všech analýz) do prvního průchodu a zadní části překladače do druhého průchodu. Tyto dva průchody si předávají data ve formě intermediálního kódu. V prvním průchodu pak jsou lexikální, syntaktický a sémantický analyzátor. Syntaktický analyzátor postupně vyžaduje na lexikálním analyzátoru symboly, pak zpracuje sám syntaxi řetězce, předá sémantickému analyzátoru, vyžádá si další symbol, ...



Obrázek 1.3: Víceprůchodový překladač, v samostatných průchodech jsou přední a zadní část překladače

Víceprůchodový překladač má tyto výhody:

- snadněji se vytváří a opravuje, lze také jednodušeji rozdělit práci mezi více programátorů,
- při překladu může být v paměti pouze ta část překladače, která zpracovává příslušný průchod, ostatní zatím nejsou v paměti zapotřebí,
- algoritmy pro optimalizaci jsou často velmi rozsáhlé a složité, pracují s delším úsekem kódu, proto mívají obvykle vlastní průchod nebo dokonce jsou rozčleněny do více průchodů.

Nevýhodou mohou být časové ztráty při ukládání dílčích výsledků překladu do pomocné paměti a jejich následném načítání a také větší spotřeba paměti.

Další informace

O víceprůchodových překladačích například na https://www.guru99.com/compiler-design-tutorial.html.

1.4 Konverzační překladače

Konverzační (interaktivní) *překladač* je takový překladač, který s programátorem během překladu komunikuje. Překlad může probíhat tak, že uživatel postupně píše řádky zdrojového kódu programu a po ukončení každého řádku je tento nový úsek předán překladači.

Konverzační překladače obvykle obsahují také příkazy *metajazyka*, které nepatří do zpracovávaného programovacího jazyka, ale jsou určeny přímo překladači. Je to například příkaz pro zjištění momentální hodnoty některé proměnné či výpisu seznamu deklarovaných proměnných, pro výpis do té doby vloženého zdrojového textu, k uložení programu, příkaz ukončující práci překladače (znamená konec vstupního zdrojového textu) atd. Konverzační překladače s přidaným grafickým rozhraním mají místo samotných metapříkazů (nebo navíc) vlastní menu, kde tyto možnosti najdeme.

Hlavní výhodou je rozšíření možností ladění a jejich zefektivnění. Jestliže syntaktický analyzátor klasického (tj. nekonverzačního) překladače objeví chybu a chce uživateli sdělit její umístění, musí tento údaj zjistit. Pak je nutné buď vypsat chybové hlášení ve znění „Došlo k chybě xxx na řádku yyy“ a nutit uživatele, aby si řádek yyy a na něm chybu xxx sám nalezl, nebo se ve vývojovém prostředí přímo vizuálně na tento řádek přenést a patřičně zvýraznit.

Konverzační překladač má obrovskou výhodu v tom, že jestliže nastane chyba, je to většinou u posledního vstupu, což bývá jeden jediný řádek. Hlášení o chybě dokonce má pro uživatele větší informační hodnotu, protože si obvykle lépe pamatuje, proč před chvílí napsal zrovna to slovo a žádné jiné, takže dokáže rychleji a lépe na chybu reagovat.

Hlavní nevýhodou je snížená možnost zpracování kontextových závislostí, sémantická struktura programu nesmí být moc složitá (například rekurzivní zpracování funkcí, dopředné definice apod. se jen těžko implementují).

Konverzační překladače (obvykle interpretační) najdeme zejména v různých výukových programech a hrách, kde uživatel zadává příkazy ve formě řetězců, ale také v textových shellech operačních systémů a v databázových systémech, tedy kdekoliv, kde zadáváme příkazy „po jednom“ na řádku.

1.5 Zpracování chyb

Pokud překladač přijde v kterékoliv fázi na chybu v zdrojovém programu, musí uživateli podat tyto informace:

- kde v programu se chyba nachází (např. číslo řádku a pozice na něm; pokud je připojen editor, tento řádek se obvykle vysvítí),
- typ chyby (např. „proměnná tohoto názvu nebyla deklarována“, „chyba v syntaxi operátoru“),
- některé překladače dokážou navrhnout možnosti nápravy chyby. Překladač by se rozhodně neměl pokoušet chyby sám opravovat bez okamžitého informování uživatele.

Zatímco u lexikální analýzy není problém kdykoliv sdělit uživateli, kde ve zdrojovém souboru k chybě došlo, u dalších fází překladu, pokud jsou umístěny v jiném průchodu, je nutné vazbu na zdro-

jový soubor vhodným způsobem vyřešit (musíme v každém okamžiku vědět, na kterém řádku a kterém znaku nebo slově řádku se momentálně nacházíme). Jedná se především o syntaktickou analýzu, protože sémantika je obvykle řešena ve stejném průchodu jako syntaxe. To můžeme udělat několika způsoby, například:

1. Součástí symbolu nebude jen jeho identifikace a sémantické atributy, ale také další dva atributy určující číslo řádku, na kterém se symbol nachází, a vzdálenost prvního znaku symbolu od začátku řádku. Tyto informace zajišťuje lexikální analyzátor.
2. Nadefinujeme speciální typ symbolu, který bude představovat přechod na nový řádek ve zdroji. Tento symbol přidá lexikální analyzátor k výstupu kdykoliv, když narazí na konec řádku ve zdroji (samozřejmě také uvnitř komentářů).

   Syntaktický analyzátor má vyhrazený čítač (celočíselnou proměnnou), který zvýší o 1, když ve svém vstupu načte symbol konce řádku, takže má přehled o tom, na kterém *řádku* zdroje se nachází. Pozici na řádku zajistíme stejně jako v případě 1, tj. uložením do atributu symbolu při lexikální analýze.


Překladač může na chybu reagovat dvěma způsoby:

- při prvním výskytu chyby se zastaví, provede diagnózu, informuje uživatele a čeká, až bude chyba opravena (například Turbo Pascal),
- pokouší se najít co nejvíce chyb najednou, zastaví se až při určitém maximálním počtu a informuje uživatele o všech objevených chybách popř. o maximálním počtu chyb, které je schopen zobrazit (například C++).


Druhý způsob využívá postup zvaný *zotavení po chybě*. Umožňuje opravit více chyb najednou bez nutnosti pokaždé znovu spouštět překladač, může však nastat situace, kdy výskyt jedné chyby ovlivní výskyt řady dalších. Typickým příkladem je překlep při deklaraci proměnné – potom všechna použití „správného“ názvu jsou považována za chybná.

Zotavení po chybě obvykle probíhá tak, že příslušný analyzátor načítá prvky ze vstupu (znaky, symboly) naprázdno bez další reakce tak dlouho, dokud se nepodaří navázat na předchozí správný průběh překladu, pak pokračuje běžným způsobem.


Chyby na straně uživatele překladače dělíme do tří kategorií:

1. Chyby související se strukturou programu (většinou lexikální nebo syntaktické), ty lze obvykle zjistit už při překladu. Této kategorii se budeme věnovat v následujících kapitolách.
2. Chyby běhové (run-time), například dělení nulou. Souvisejí obvykle s momentální hodnotou proměnných a lze je jen těžko zjistit (některé překladače při zjištění možnosti run-time chyby generují varování – warning).
3. Chyby logické (chybná posloupnost příkazů, záměna operátorů, překlep v čísle apod.), které při překladu prakticky nelze odhalit.

1.6 Překladače ve vztahu k jiným programům

1.6.1 Generátory překladačů


Překladače můžeme psát v Assembleru (nejefektivnější, ale také nejnáročnější) nebo ve vyšších programovacích jazycích, ale dnes existují také speciální programy nazývané *generátory překladačů*,

překladače kompilátorů nebo *systémy pro psaní překladačů*. Tyto systémy vyžadují na svém vstupu specifikaci zdrojového jazyka, tedy vlastně lexikální a syntaktickou strukturu. Další důležitou informací je popis výstupu překladače, kde určíme, pro jaký typ počítače a operačního systému má být kód generován.

Každý překladač je charakterizován třemi jazyky:

- jazyk, ve kterém je sám napsán,
- zdrojový jazyk, který přijímá,
- cílový jazyk, ve kterém je jeho výstup.

Z programů pro generování překladačů (resp. jejich částí) jsou známé např. Lex nebo Flex (generují lexikální analyzátor) a Yacc nebo Bison (syntaktický analyzátor)[2]

Další informace

Stručně o nástrojích pro generování překladačů:
https://www.geeksforgeeks.org/compiler-construction-tools/

1.6.2 Aplikace pro jinou platformu

Kompilátor může pracovat na jednom počítači a generovat programy v cílovém jazyce pro úplně jiný počítač (myšleno pro jinou hardwarovou nebo softwarovou platformu). Je sice nevýhodou, že vygenerovaný program nelze ihned po přeložení přímo spustit (je psán pro jiný počítač, než na kterém byl přeložen), ale tento postup značně ulehčuje práci programátorům, kteří si nemusejí pro každou zakázku pořizovat specifický hardware a software. Takové řešení je obvyklé především tam, kde by se na cílové platformě špatně programovalo, například u programů pro malá mobilní zařízení (mobilní telefony, zařízení internetu věcí), herní konzole nebo roboty.

Dnes se běžně problémy překladu pro jinou platformu řeší použitím emulátorů. *Emulátor* je program, který simuluje prostředí jiného počítače nebo operačního systému, a tedy umožňuje spouštění aplikací, které by jinak nebylo možné na daném počítači, resp. operačním systému, spustit. Pokud chceme programovat pro jiné zařízení než to, na kterém pracujeme, obvykle používáme určité vývojové prostředí, jehož součástí už emulátor bývá.

1.6.3 Portování

S překladači úzce souvisí pojem *portování*. Jde o proces přenesení operačního systému nebo programu na jinou platformu, v případě operačních systémů hardwarovou – jiný typ počítače (především procesoru, s jinou instrukční sadou), v případě ostatních programů spíše softwarovou (na jiný operační systém) nebo se změna musí týkat hardwarové i softwarové platformy (ovladače nebo jakékoliv programy psané v nižším programovacím jazyce). Může jít i o nutnost provedení změn přímo ve zdrojovém kódu, nejen samotný překlad.

Tento pojem se často používá v souvislosti s UNIXem a Linuxem, protože varianty těchto operačních systémů, na rozdíl např. od MS Windows, dnes běží téměř na čemkoliv. UNIX byl zpočátku určen pro počítač PDP-7, ale programován byl na úplně jiném počítači a pro přenos na PDP-7 bylo nutné

[2] Jsou k dosažení jako freeware na Internetu, lze také zakoupit licence těchto programů v propracovanějších verzích.

portování. Linux dnes najdeme nejen na strojích kompatibilních s procesory Intel, ale také PowerPC, Alpha, Sparc, clustery v datových centrech atd., varianta Linuxu pro 64-bitové počítače také existovala výrazně dříve než varianta MS Windows.

Portování je také forma překladu. Vstupem bývá obvykle zdrojový kód překládaného programu, výstupem je kód pro jinou platformu, a to buď zdrojový nebo přímo cílový (zdrojový se po případných úpravách přeloží překladačem napsaným přímo pro cílovou platformu). Proto hodně záleží na typu zdrojového kódu. Obecně platí, že vyšší programovací jazyk se portuje jednodušeji, protože programovací jazyky nižší úrovně včetně Assembleru jsou příliš hardwarově závislé. Z tohoto důvodu byly zdrojové kódy operačního systému UNIX brzy po svém vzniku přepsány do jazyka C, speciálně pro tento účel vytvořeného.

Portovat se dají nejen operační systémy, ale také samozřejmě jakékoliv další programy. Důvodem je nejen hardwarová, ale také softwarová kompatibilita (aby běžely na určitém operačním systému a mohly využívat jeho služeb). Protože však se dnes pro jejich tvorbu používají převážně vyšší programovací jazyky, ve většině případů nemá tento proces příliš smysl (zdrojový program např. v jazyce C je přenositelný a přeložitelný do spustitelného souboru v různých operačních systémech bez jakýchkoliv úprav[3]). Výjimkou jsou programy, ve kterých je závislost na hardwaru nebo operačním systému nutností (například ovladače).

V případě interpretovaných jazyků obvykle ani není třeba při změně platformy provádět změny v kódu, s přenositelností se u nich automaticky počítá. Ovšem pro cílovou platformu musí existovat interpretační program.

1.7 Editory pro překladače

Většina překladačů je dodávána s editorem zdrojového jazyka, z kterého lze volat programy pro překlad či ladění zdrojového programu. Běžně se také dají sehnat také editory „externí“ od třetích stran, které spolupracují s několika běžnými programovacími jazyky a dokážou zvýrazňovat jejich syntaxi. Často se jedná o freeware dostupný na Internetu nebo open-source software (například ve Windows se často používá PSPad, v Linuxu vim, emacs, kate, kile a další).

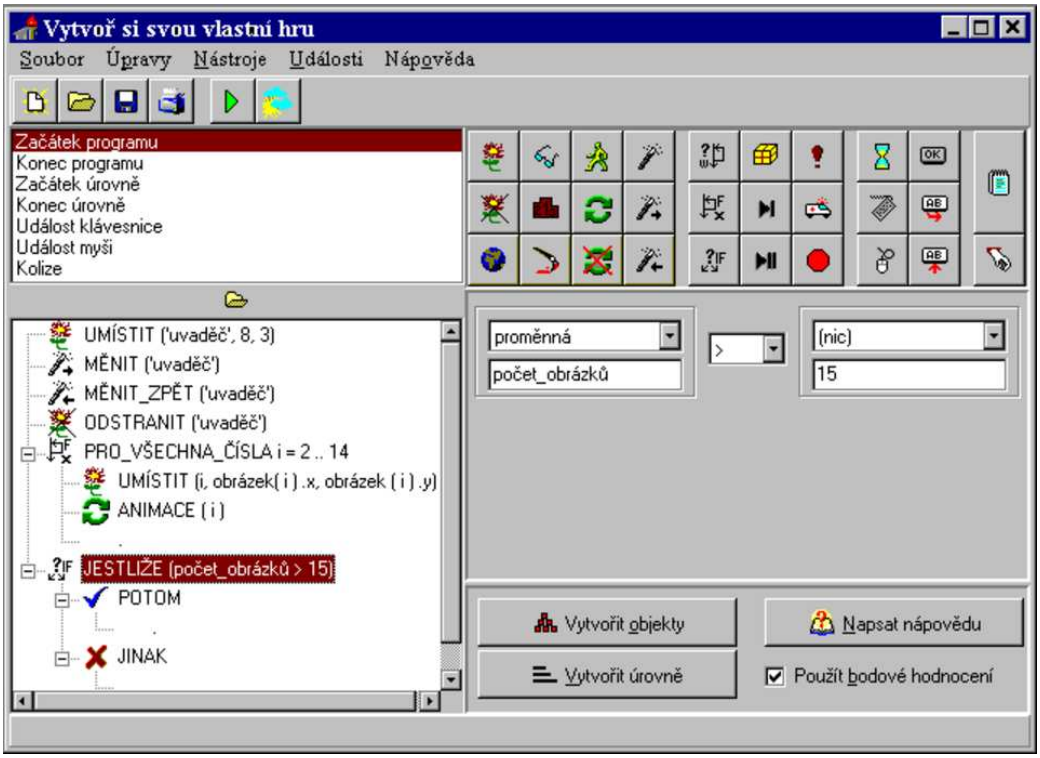
Editor dodávaný s překladačem bývá zpravidla napsán v zdrojovém jazyce, který přijímá jeho překladač (a také bývá tímto překladačem přeložen), aby autor demonstroval použitelnost jazyka a překladače.

Tyto editory mohou být realizovány několika způsoby:

Textový editor. Programovat můžeme v jednoduchých editorech neukládajících formátování. Pod Windows jsou oblíbené například PSPad nebo Notepad++, v Linuxu vim, emacs, kate a další součásti standardních distribucí. V těchto nástrojích se celkem pohodlně píšou skripty nebo kód webových stránek, umějí vysvítit syntaxi pro konkrétní programovací či značkovací jazyk.

Grafický editor. Takovéto editory se používají pro velmi jednoduché programovací jazyky, případně hry nebo rozhraní pro vytváření her. Na obrázku 1.4 je ukázka editoru na vytváření jednoduchých událostmi řízených her určeného pro děti (autor právě tvoří scénu na začátku hry, kdy se na jevišti objeví uvaděč, ukloní se a zmizí a potom se spustí animace několika obrázků – obrázky mohou

[3] Programy pro UNIX a Linux ostatně bývají často dodávány ve zdrojovém tvaru v jazyce C nebo jiném, uživatel si je pomocí utilit dodávaných s operačním systémem přeloží a nemá problémy s kompatibilitou. Obvyklý sled programů spouštěných pro překlad je `./configure ; make ; sudo make install`.



Obrázek 1.4: Ukázka prostředí grafického editoru určeného pro děti

být reprezentovány názvem nebo jejich pořadovým číslem, obrázek, který může být animován, obsahuje ve skutečnosti několik obrázků, které se v krátkých intervalech střídají).

Příkazy jsou zobrazeny ve stromové struktuře podobně jak je běžné u struktury adresářů (složek). Každý uzel stromu představuje jeden příkaz. Každá funkce včetně hlavního programu má svůj vlastní strom, u složených příkazů rozhodování, cyklů a složených parametrů funkcí má uzel jeden nebo více podřízených uzlů. Uzlům mohou být přiřazovány ikony pro snadnější rozlišení jejich významu. Tato struktura značně usnadňuje lexikální a syntaktickou analýzu, obojí lze provádět již při vytváření programu (obvykle se údaje zadávají pomocí zvláštního dialogového okna).

Strukturogram. Tento způsob prezentace zdrojového kódu se používal již v počátcích programování. Forma a tvar jednotlivých prvků struktury závisí jen na autorovi. Může to být na papíře načmáraná struktura programu pomocí vývojových diagramů, ale také elektronická podoba vytvářená v některém programu. Spočívá v postupném vytváření struktury podobné n-árnímu stromu, která se vyhodnocuje shora dolů a zleva doprava.

Velmi oblíbeným jazykem umožňujícím strukturované programování s „grafickou“ podporou je Scratch, kde se dá programovat jednoduše přesouváním bloků. Na podobném principu je založeno také grafické programovací prostředí MakeCode pro platformu micro:bit.

O něco starší, ale stále živý projekt je SGP, informace na https://sgpsys.com/. Známým programem z projektu SGP je především *Baltazar*, jeho zjednodušenou verzi *Baltík* řadíme spíše do předchozí skupiny – grafických editorů.

Vývojové prostředí. Tyto „editory“ jsou nyní nejpoužívanější (Microsoft Visual Studio, Embarcadero C++ Builder, Dev-C++, Embarcadero Delphi, pro Linux QtDesigner nebo KDevelop, ...). Grafické prostředí umožňuje jednoduše vytvářet a umísťovat objekty (např. obrázek, textové pole,

tlačítko) a zadávat jejich vlastnosti, zatímco textová část editoru slouží k psaní procedur a funkcí, manipulaci s objekty a samotnému programování. V textové části se prosazují nové vlastnosti zjednodušující práci programátorům, například skrývání podřízených bloků kódu. Již delší dobu se tento typ editorů prosazuje také při tvorbě webových prezentací (tzv. WYSIWYG editory). V oblasti dětských programů pro výuku programování by se snad do této kategorie daly zařadit některé projekty pracující s „robotem Karlem“ a mnohé jmenované v předchozím bodu.

Další informace

Zajímavý přehled volně šiřitelných nástrojů pro programování je na
https://bastlirna.hwkitchen.cz/top-11-free-nastroju-programovani-pro-deti/

Toto je jen přehled nejpoužívanějších technik. Samotný editor lze implementovat mnoha způsoby – vybíráme především podle rozsáhlosti a složitosti syntaxe zdrojového jazyka a také podle toho, jakému uživateli je editor určen. Uživatele editoru můžeme rozdělit do tří skupin:

- *Profesionální programátor* vyžaduje, aby všechny potřebné nástroje byly rychle přístupné a aby nebyl zbytečně zdržován pokusy editoru „napovídat“ (i když například dokončování syntaxe se občas hodí). Nejvhodnější je kombinace textového a grafického editoru ve vývojovém prostředí s tím, že důležitější je textová část, a grafická část se používá pouze jako doplněk pro zrychlení některých operací[4]. Textový editor by rozhodně měl barevně vyznačovat syntaxi, alespoň klíčová slova. Nápověda by se měla soustředit především na syntaxi a sémantiku příkazů (název příkazu, typ a pořadí jeho parametrů, ...).
- *Programátor–začátečník* potřebuje především interaktivní a rozsáhlou nápovědu. Prostředí by mělo být orientováno více graficky, nezáleží ani tak na rychlosti ovládání, jako spíše na snadnosti nalezení příslušného nástroje. Textová část editoru má barevně vyznačovat syntaxi, případně včetně řetězců znaků, které překladač považuje za chybné (provádí lexikální analýzu již během vytváření zdrojového programu nebo jeho načítání z paměťového média). Nápověda by se neměla omezovat pouze na to, jak jednotlivé příkazy vypadají a jaké parametry vyžadují, ale také na to, jaké možnosti jazyk nabízí, jak co naprogramovat, kde čekají různá úskalí, co by mělo předcházet použití daného příkazu, a to vše nejlépe doprovodit příklady.
- *Dítě* chápe programování především jako hru, proto je vhodné, když editor připomíná prostředí jednoduchých počítačových her. Prostředí pro malé děti by mělo být spíše grafické s textovou částí jen tam, kde je to bezpodmínečně nutné, barevné, nemělo by nutit k častému používání klávesnice. Pro větší děti je již možné rozšířit funkci textové části editoru. Nápověda by měla být konstruována s ohledem na věk uživatele, tedy interaktivně a bez používání mnoha odborných termínů. Její důležitou součástí jsou příklady a vzorová řešení.

Úkoly

1. U následujících (většinou interpretovaných) programovacích jazyků zjistěte
   - základní informace o tomto jazyce (použijte Internet) – typ jazyka, pro jaké softwarové platformy je určen, jak se zachází s datovými typy, některé základní příkazy,

[4] RAD – Rapid Application Development, rychlý vývoj aplikací, je trend pro vývojová prostředí, kdy alespoň část GUI vyvíjené aplikace programátor určuje rychle „pomocí myši“.

- zda pro něj existuje možnost vygenerovat cílový kód a jakým způsobem se to provádí (případně zjistěte volně dostupné překladače[5]).

Flex	Lisp	Perl	SmallTalk
Goedel	Logo	Prolog	Tcl
Haskell	Lua	Python	Tcl/Tk
Java	Mercury	Ruby	

2. Zjistěte, jakým způsobem pracuje program gcc pro překlad zdrojových souborů některých programovacích jazyků v Linuxu. Zobrazte manuálovou stránku se seznamem přepínačů tohoto programu a zjistěte, který přepínač je třeba použít, pokud chcete zadat název výstupního souboru. Vyzkoušejte na jednoduchém programu typu „Hello world“.
3. Zjistěte, zda je možné používat některý UNIXový textový shell v emulovaném prostředí UNIXu pod Windows (například v prostředí Cygwin).

[5] Jedním z nejlepších zdrojů překladačů je například https://www.thefreecountry.com/, a samozřejmě https://www.google.com/, kde do vyhledávacího pole zadáme název programovacího jazyka. Mnohé z těchto jazyků jsou standardně nainstalovány v Linuxu (nebo není problém je běžným způsobem doinstalovat z repozitářů), včetně příslušných manuálových stránek.

Kapitola 2

Lexikální analýza

Rychlý náhled: V této kapitole se budeme zabývat první fází zpracování zdrojového programu, kterou je lexikální analýza. Využijeme zde poznatky teoretické informatiky, která nám nabízí jednoduché prostředky pro popis lexikální struktury zdrojového jazyka (regulární gramatiky) a pro určení postupu samotné analýzy (konečné automaty). Ukážeme si také, jak jednoduše takto reprezentovaný postup naprogramovat.

Klíčová slova: Lexikální analýza, symbol, lexém, regulární gramatika, konečný automat, stavové programování.

Cíle studia: Cílem této kapitoly je naučit se navrhnout lexikální strukturu zvoleného jazyka a naprogramovat jeho lexikální analýzu.

2.1 Popis lexikální struktury jazyka

V následující tabulce jsou shrnuty nejdůležitější vlastnosti lexikální analýzy.

Vstup:	*zdrojový program překladače*
Výstup:	*posloupnost symbolů*
Lexikální chyby:	*v rámci jednoho symbolu, například posloupnost znaků, která není symbolem (72R4), znak nepatřící do abecedy jazyka, . . .*

Tabulka 2.1: Vlastnosti lexikální analýzy

Vstup lexikálního analyzátoru může být samozřejmě různý, záleží na tom, s jakým typem editoru počítáme. Lexikální analyzátor se také dá naprogramovat tak, aby dokázal přijímat více různých vstupních formátů, ale to bývá řešeno jednoduše konverzními programy převádějícími jeden vstupní formát na druhý.

Dřív než se pustíme do návrhu lexikálního analyzátoru, měli bychom si ujasnit, v jakém formátu bude jeho vstup, tedy jaký typ dat bude zpracovávat.

Obvykle se používají tyto vstupní formáty:

- *text* (většinou jeden nebo několik textových souborů),
- *binární formát* (generují některé grafické editory, může zachycovat např. strukturu graficky nadefinovaného formuláře a jiné prvky, které uživatel „umístil myší“),
- *vázaný text* (vyžaduje předem danou strukturu – např. každý příkaz na novém řádku), částečné vázání je hodně oblíbené v modernějších interpretovaných jazycích, kde každý příkaz je na samostatném řádku a závorky ohraničující blok příkazů jsou nahrazeny velikostí odsazení bloku zleva (například v Pythonu),
- *dynamická struktura v paměti* (popř. se lexikální analyzátor podílí na jejím vytváření).

Úkolem lexikálního analyzátoru je převést zdrojový program na posloupnost nejmenších částí s vlastním významem. Tyto části nazýváme *symboly* (také atomy, lexémy, lexikální jednotky, ...). Symbolem může být například číslo, klíčové slovo, název proměnné, aritmetický operátor pro sčítání, relační operátor „menší-rovno“ apod. Symbol má dvě základní části:

- identifikace (typ) – o jaký typ symbolu jde,
- atribut (-y) – skutečná hodnota čísla, název proměnné, pozice ve zdrojovém souboru, apod.

Příklad

Podíváme se na jednoduchý program v jazyce pracujícím s celými čísly a výstup pro tento program generovaný lexikálním analyzátorem. Vstup je následující:

```
CONST hodn = 32;
VAR prom;
BEGIN
  prom := 25 * (hodn + 4);
  IF prom < 100 THEN PRINT prom
                ELSE PRINT prom - 100;
END
```

Výstup lexikálního analyzátoru je tento soubor:

```
S_CONST
S_ID    HODN
S_EQ
S_NUM    32
S_SEM
S_VAR
S_ID    PROM
S_SEM
S_BEGIN
S_ID    PROM
S_IS
S_NUM    25
S_MUL
S_LPAR
S_ID     HODN
S_PLUS
S_NUM    4
...
S_NUM    100
S_SEM
S_END
```

Identifikaci symbolů můžeme stanovit jinak, například všechny operátory budou mít společný identifikátor `S_OPERATOR` a odlišnou část s atributy. To však nemusí být zrovna nejvhodnější řešení, protože v následujících fázích se se symboly hůře pracuje, třeba při určování priority operátorů.

Jiný může být také tvar výstupu. Mohli bychom výslednou posloupnost symbolů uložit do textového souboru, ovšem další fáze by byla nucena opět pracovat s textovými znaky a znovu bychom museli načítat písmeno po písmenu. To není moc efektivní. Tento typ výstupu používáme zpravidla jen ve fázi ladění analyzátoru, v textovém výstupu se snadněji hledají chyby.

Výstupem může být také binární soubor (symboly se z binárního souboru načítají jednodušeji než z textového), pole či dynamický seznam záznamů využívajících nadefinovaný výčtový typ pro identifikaci symbolu (atribut může být řetězec nebo třeba variantní záznam – union).

Při stanovení lexikální struktury jazyka začínáme na čistě abstraktní bázi – určujeme, jaká bude abeceda jazyka a jaké typy symbolů se v jazyce mohou vyskytovat, a zda bude „case-sensitive“, tedy jestli budeme rozlišovat malá a velká písmena.

Příklad

Abeceda: $\Sigma = \{A, \ldots, Z, a, \ldots, z, 0, \ldots, 9, +, -, *, /, >, <, =, (,), ;, :\}$
Symboly:

- celá nezáporná čísla (pro konstanty),
- rezervované identifikátory – klíčová slova: `BEGIN`, `END`, `VAR`, `CONST`, `IF`, `THEN`, `ELSE`, `PRINT`,
- ostatní identifikátory – pro názvy proměnných,
- aritmetické operátory (+, −, ∗, /),
- relační operátory (<, <=, >, >=, <>, =),
- operátor přiřazení (:=),
- pomocné symboly (závorky, středník).

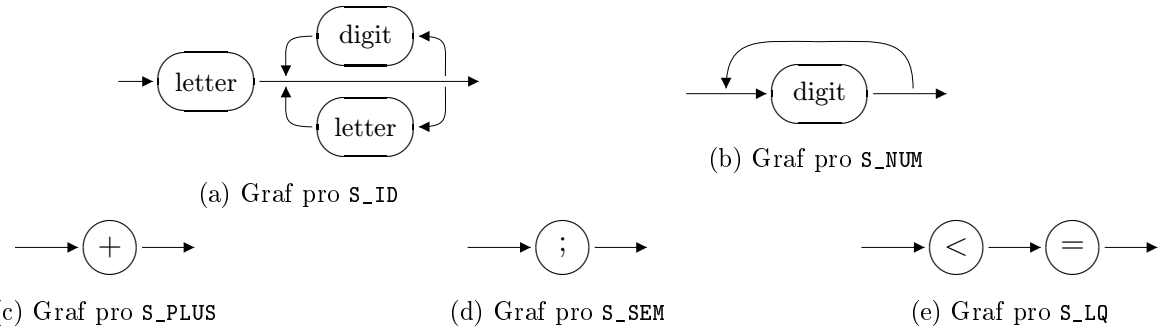
Z abstraktní báze se posunujeme ke konkrétní reprezentaci struktury symbolů. Můžeme použít *syntaktické grafy* nebo přímo pravidla regulární gramatiky.

Příklad

Nadefinujeme pomocí syntaktických grafů identifikátory (`S_ID` – zahrnuje klíčová slova a názvy proměnných), čísla, aritmetický operátor pro sčítání (`S_PLUS`), symbol pro středník (`S_SEM`) a relační operátor „menší-rovno“ (`S_LQ`) z příkladu 2.1. Klíčová slova zatím nebudeme odlišovat od ostatních identifikátorů.

Terminál *letter* označuje jakékoliv písmeno z množiny {A,...,Z,a,...,z}, terminál *digit* jakoukoliv číslici z množiny {0,...,9}. Asi nejsložitější je syntaktický graf na obrázku 2.1a. Na grafu vidíme, že identifikátor musí začínat písmenem (vždy alespoň jedno písmeno), a pak mohou následovat písmena (v grafu návrat směrem dolů) nebo číslice (návrat směrem nahoru).

Sestavíme gramatiku popisující jazyk naznačený v tomto příkladu.
$G = (N, T, P, S)$, $T = \Sigma$ (případně můžeme přidat symbol pro mezeru a konec řádku),
$N = \{S, A, B, C, D, E\}$, P obsahuje pravidla (l je *letter* – písmeno, d je *digit* – číslice):



(a) Graf pro `S_ID`

(b) Graf pro `S_NUM`

(c) Graf pro `S_PLUS`

(d) Graf pro `S_SEM`

(e) Graf pro `S_LQ`

Obrázek 2.1: Syntaktické grafy některých symbolů

S	$\rightarrow$	$l \mid lA$	identifikátory
A	$\rightarrow$	$l \mid d \mid lA \mid dA$	
S	$\rightarrow$	$d \mid dB$	čísla
B	$\rightarrow$	$d \mid dB$	
S	$\rightarrow$	$+ \mid - \mid * \mid /$	aritmetické operátory
S	$\rightarrow$	$> \mid < \mid = \mid <C \mid >D$	relační operátory
C	$\rightarrow$	$= \mid >$	
D	$\rightarrow$	$=$	
S	$\rightarrow$	$:E$	operátor přiřazení
E	$\rightarrow$	$=$	
S	$\rightarrow$	$(\mid) \mid ;$	pomocné symboly
S	$\rightarrow$	$mezera \mid konec\_řádku$	

Všimněte si, že ze startovacího symbolu S vygenerujeme vždy právě jeden symbol. Pro další symbol se musíme opět vrátit k symbolu S.

2.2 Rozpoznávání symbolů

V předchozí kapitole jsme určili lexikální strukturu jazyka pomocí regulární gramatiky. Gramatika dokáže jazyk popsat, ale pokud chceme zjistit, zda zadané slovo patří do jazyka (tj. určit ze vstupního řetězce, o jaký symbol jde), potřebujeme konečný automat, který bude pracovat takto:

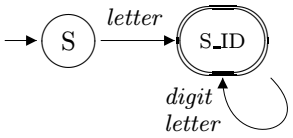
1. Na vstupu máme řetězec znaků, který chceme analyzovat.
2. Automat postupně čte znaky ze vstupu, mění svůj stav, a pokud je to nutné, načtené znaky ukládá na výstupní pásku.
3. Pro každý typ symbolu má automat jiný koncový stav. Podle toho, ve kterém stavu ukončí výpočet, určíme, o jaký symbol se jedná.

Kdybychom pro rozpoznávání symbolů na vstupu použili „klasické“ funkce typu `strcmp`, dostaneme se především u rozsáhlejšího překladače do velkých problémů: při každém porovnávání bychom se totiž opakovaně vraceli na začátek symbolu, tentýž znak bychom museli opakovaně zpracovávat hlavně při

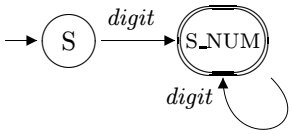
rozpoznávání klíčových slov. Konečný automat má výhodu v tom, že každý znak zpracuje opravdu jen jednou.

Příklad

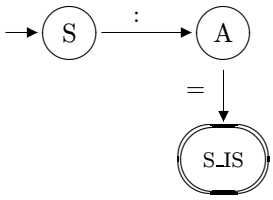
Podle regulární gramatiky v příkladu 2.1 sestrojíme konečný automat. Ukážeme opět jen části pro rozpoznání několika symbolů. Řešení pro reprezentaci čísel, identifikátorů, symbolu přiřazení a některých relačních operátorů najdeme na obrázku 2.2.



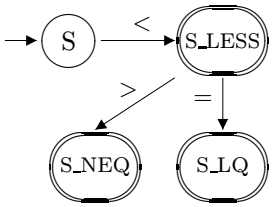
(a) Konečný automat pro `S_ID`



(b) Konečný automat pro `S_NUM`



(c) Konečný automat pro symbol „:=“



(d) Konečný automat pro „<=“, „<“ a „<>“

Obrázek 2.2: Konečný automat pro některé symboly

Ostatní diagramy jsou podobné, jejich vytvoření necháváme na čtenáři.

Všechny tyto stavové diagramy popisují konečné deterministické automaty, jejichž koncové stavy představují symboly. Když vytvoříme stavové diagramy pro všechny symboly a shrneme je (tj. sloučíme počáteční stavy S stavových diagramů pro všechna slova jazyka), získáme konečný deterministický automat rozpoznávající jazyk z příkladů.

2.3 Implementace

Při programování překladače je velmi důležitá *volba programovacího jazyka*, ve kterém budeme pracovat. Existuje mnoho programovacích jazyků dostatečně silných pro psaní překladačů, každý z nich má své výhody a nevýhody. Obecně platí, že jazyky vycházející z Pascalu (včetně Delphi) jsou výhodné především pro jednoduchost práce s množinami znaků, jazyky vycházející z C a C++ jsou považovány za silnější (robustnější) a pokročilí programátoři jsou obvykle na tyto jazyky zvyklí. Navíc nebývá problém s portováním na jinou platformu, norma se nijak divoce nemění (tudíž kód bude znovupoužitelný po mnoho let) a je velký výběr ve vývojových prostředích. Java a C# jsou taktéž použitelné, jen v Javě se musíme přizpůsobit čistě objektovému návrhu a C# je zase poněkud těžkopádný.

Pokud píšeme interpretační překladač, není až takový problém psát v některém skriptovacím jazyce, ale musíme se smířit s tím, že nebudeme mít až takovou kontrolu nad paměťovým prostorem (což není dobré z pohledu optimality běhu programu), a také – skriptovací jazyky moc s tímto způsobem využití nepočítají...

V předchozí sekci jsme vytvořili konečný automat rozpoznávající zdrojový jazyk překladače. Nyní sestavíme program, který realizuje výpočet tohoto automatu.

Budeme postupovat takto:

1. V každém stavu automatu program načte ze zdrojového souboru jeden znak a podle něho se rozhodne, kterou větví pokračovat.
2. V koncových stavech je třeba provést test, zda je načtený symbol korektně ukončen, tedy načteme následující znak.
3. Pokud automat nenalezne větev, po které by pokračoval a je v koncovém stavu, právě načetl jeden celý symbol a po analýze dalšího znaku (viz předchozí bod) se přesouvá do počátečního stavu S, aby (po případné přestávce) mohl načítat další symbol.
4. Pokud automat nenalezne větev, po které by pokračoval a nenachází se v koncovém stavu, potom načtený znak je chybný, došlo k lexikální chybě.

V následujících sekcích probereme jednotlivé části lexikálního analyzátoru, celý kód zde není vcelku uveden a naprogramování některých jednodušších funkcí necháváme na čtenáři. Kód se týká jazyka z předchozích příkladů, pokud není uvedeno jinak.

2.3.1 Vstup a výstup lexikálního analyzátoru

Nadále předpokládáme, že lexikální a syntaktický analyzátor se nacházejí v jednom průchodu. Proto lexikální analyzátor implementujeme jako funkci, která jako výsledek své práce vrátí v proměnné jeden symbol, a budeme počítat s tím, že syntaktický analyzátor tuto funkci průběžně volá, kdykoliv potřebuje další symbol.

Nejdřív vytvoříme výčtový typ představující názvy všech používaných symbolů. Tato data nám budou sloužit ke zjednodušení tvaru výstupu – místo řetězce představujícího název symbolu pracujeme pouze s indexem zabírajícím jeden nebo dva Byte.

```
enum TTypSymbolu { S_NOTHING, S_ENDOFFILE, S_SEM, S_LPAR, S_RPAR,
  S_BEGIN, S_END, S_CONST, S_VAR, S_IF, S_THEN, S_ELSE, S_PRINT,
  S_ID, S_NUM, S_IS, S_PLUS, S_MINUS, S_MUL, S_DIV,
  S_EQ, S_NEQ, S_LESS, S_GRT, S_LQ, S_GQ };

struct TSymbol {
  TTypSymbolu typ;
  string atrib;
};
```

V našem případě bude výstupem každého volání funkce lexikálního analyzátoru pouze jeden symbol, který můžeme uložit do globální proměnné nebo předat jako parametr či návratovou hodnotu funkce.

Pokud první dvě fáze rozdělíme do různých průchodů, použijeme soubor, stream, dynamický seznam či podobnou datovou strukturu pro posloupnost symbolů reprezentující celý vstup. Výstupní soubor může být textový nebo také binární s tím, že lze ukládat symboly v optimálnějším formátu (identifikace symbolu je reprezentována číslem podle pozice ve výčtovém typu, hodnoty datového typu číslo jako čísla v jednom nebo více Bytech apod.).

Atribut symbolu reprezentujeme řetězcem tak, jak je použito výše, nebo třeba variantním záznamem (unionem), ve kterém už lexikální analýza odliší různé datové typy jazyka a není tím zatěžován syntaktický analyzátor. Navíc vnitřní reprezentace například běžného čísla v binárním tvaru (integer)

zabere méně paměti než ve tvaru textovém (s použitím znaků `'0','1',...,'9'`) a odpadají další konverze.

 Práci se vstupem můžeme řešit různými způsoby – můžeme načítat znak po znaku, nebo třeba načíst celý řádek do řetězce, a pak se v něm posouvat když dojdeme na konec, načteme další řádek). Vpodstatě to je jedno. Dříve byl první způsob považován za neoptimální, protože znamenal neustálé pomalé přístupy na pevné datové médium, ale dnes se při otevření souboru vytvoří v paměti stream s celým obsahem souboru a není není nutné tahat data z disku při každém přístupu do souboru.

Pro uschování načtené části vstupu zvolíme tuto strukturu:

```
struct TVstup {
  char znak;         // zpracovávaný řádek
  int cisloRad;      // číslo řádku ve vstupním souboru
  int pozice;        // pozice posledního načteného znaku na řádku
  bool konec;        // zde lexikální analyzátor indikuje, že celý vstup byl načten
};

TSymbol symbol;              // právě zpracovávaný symbol načítaný lex. analyzátorem
string nazevsouboru;         // název zdrojového (vstupního) souboru
FILE *soubor;                // otevřený zdrojový (vstupní) soubor
TVstup vstup;                // proměnná pro právě načtený znak ze vstupního souboru
... // inicializace, otevření vstupního souboru, atd.

// Následující funkci zavoláme vždy, když při lexikální analýze potřebujeme další
    znak:
char dejZnak() {
  while (1) {
    if (feof(soubor)) {           // konec souboru
      vstup.konec = true;
      vstup.znak = '\0';
      break;
    }
    vstup.znak = fgetc(soubor);
    if (vstup.znak == '\n') {     // konec řádku
      vstup.cisloRad++;
      vstup.pozice = 0;
      break;
    }
    else {                        // ani konec souboru, ani konec řádku
      vstup.znak = toupper(vstup.znak);      // převod na velké písmeno
      vstup.pozice++;
      break;
    }
  }
  return vstup.znak;
}
```

„Aktivní“ – právě zpracovávaný – znak je ve vnitřní proměnné `vstup.znak`. Předpokládáme, že je načtena knihovna `stdio.h`, přepis na `iostream` nebo jinou podobnou knihovnu studenti určitě zvládnou.

Proměnnou `vstup.cisloRad` zachycující číslo zpracovávaného řádku zdrojového souboru použijeme především při výskytu lexikální chyby. Tuto informaci také můžeme ve vhodné formě předat dalším částem překladače (například jako další atribut symbolu nebo nový speciální typ symbolu), aby bylo kdykoliv možné zjistit, na kterém řádku zdrojového souboru se chyba nachází (to má smysl obvykle v případě, že fáze lexikální analýzy je v samostatném průchodu). Pozice na načteném řádku pro bližší určení chyby je v proměnné `vstup.pozice`.

Na konci souboru vrací funkce v proměnné `vstup.znak` znak s kódem 0. Pro jednoduchost náš překladač nebude rozlišovat velká a malá písmena, proto do funkce zahrneme převod malých písmen na velká.

Pro skutečný zdrojový jazyk bude implementace složitější, například funkce by měla automaticky vynechávat komentáře (ale přesto je zahrnovat do počtu řádků, aby bylo možné podle čísla řádku lokalizovat případnou chybu ve zdroji).

2.3.2 Základní metody

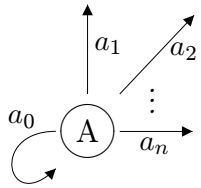
Naším úkolem je přepsat konečný automat na program. Zde zohledňujeme především to, o jaký jazyk se jedná. Metody pro přepis konečného automatu na program můžeme rozdělit do dvou skupin:

- implementace vhodné zejména pro nekonečné jazyky, kdy se hůře rozlišují klíčová slova od proměnných, ale jednodušší symboly se rozlišují naopak velmi efektivně,
- implementace vhodné především pro konečné jazyky obsahující symboly reprezentované ve zdrojovém programu delšími řetězci (klíčová slova, proměnné).

Ukazuje se, že výhodou může být zkombinování obou typů implementací, a to tak, že nejdříve načteme symbol metodou z první skupiny (zatím nerozlišujeme mezi klíčovými slovy a jinými identifikátory), a pokud je to identifikátor, použijeme na něj některou z metod druhé skupiny.

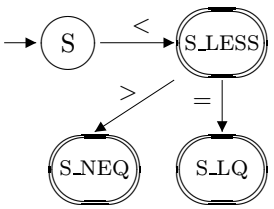
A) Přímé stavové programování

Každý stav automatu přepisujeme takto: pro reprezentaci smyčky přes jeden stav použijeme příkaz `while`, ostatní rozlišíme příkazem `switch` (`case`).




```
while (zn == a0) {   // mezery, komentáře, název proměnné,...
  zn = dejZnak();
  ...
}
switch (zn) {
  case a1: ... break;
  case a2: ... break;
  ...
  default: vypis_chybu(...);
}
```

Například podle obrázku 2.2d na straně 19 (je uveden také níže) postupujeme následovně (dále budeme používat proměnnou `vstup` podle kódu na předchozí straně):




```
while (!vstup.konec && isspace(vstup.znak)) {
  dejZnak();        // posun na vstupu, znak do vstup.znak
}
switch (vstup.znak) {
  ...
  case '<':
    dejZnak();     // potřebujeme vědět, co následuje
    switch (vstup.znak) {
      case '>': symbol.typ = S_NEQ; dejZnak(); break;
      case '=': symbol.typ = S_LQ;  dejZnak(); break;
      default: symbol.typ = S_LESS;
    }
    ...
  default: vypis_chybu(...);  // znak, který zde nemá být
}
```

Metoda přímého stavového programování (stav reprezentován místem v programu) je určena pro nekonečné jazyky. U této metody jsme omezeni pouze podmínkou, aby se v automatu *nenacházela smyčka přes více než jeden stav* (smyčku přes jeden stav, tedy začínající a končící v tomtéž stavu a neprocházející jinými, dokážeme zachytit příkazem cyklu).

B) Tabulka přechodů jako celočíselná matice

Pro gramatiku sestavíme deterministickou tabulku přechodů konečného automatu. Pokud je automat nedeterministický, upravíme na deterministický automat, lze ho však obvykle navrhnout již jako deterministický.

Nejdřív sestavíme gramatiku. Gramatika musí být regulární, tedy pravidla mají tvar $A \to aB$ nebo $A \to a$, kde A, B jsou neterminály, a je terminální symbol. Aby bylo vytvoření tabulky přechodů podle gramatiky co nejjednodušší, použijeme pro neterminály (kromě startovacího symbolu gramatiky) indexování čísly – indexy budou odpovídat stavům konečného automatu reprezentovaného tabulkou. Tabulka přechodů bude představovat matici, jejíž řádky jsou ohodnoceny čísly přiřazenými stavům, sloupce znamenají jednotlivé terminální symboly jazyka. Při výpočtu přecházíme mezi stavy tak, že se pohybujeme v této matici.

Příklad

Je dán konečný jazyk $L = \{\text{if, then, else, this}\}$. Sestrojíme gramatiku, podle ní tabulku přechodů a tu naprogramujeme.

$G = (N, T, P, S)$, kde $N = \{S, A_1, A_2, \ldots, A_8\}$, T $= \{i, f, t, h, e, n, l, s\}$

$S \to iA_1$
$A_1 \to f$
$[\Rightarrow IF]$

$S \to tA_2$
$A_2 \to hA_3$
$A_3 \to eA_4 \mid iA_8$
$A_4 \to n$
$[\Rightarrow THEN]$

$A_8 \to s$
$[\Rightarrow THIS]$

$S \to eA_5$
$A_5 \to lA_6$
$A_6 \to sA_7$
$A_7 \to e$
$[\Rightarrow ELSE]$

Nyní podle gramatiky sestavíme tabulku přechodů. Stavy $0, \ldots, 8$ přejmeme z gramatiky (0 odpovídá S), budeme potřebovat další stavy:

9 ... chybový stav
10 ... načteno *if*
11 ... načteno *then*
12 ... načteno *else*
13 ... načteno *this*

V prázdných buňkách je číslo 9, tedy chybový stav. Jde o konečný jazyk, v koncových stavech a při chybě končí výpočet, proto spodní část tabulky od řádku 9 vlastně nepotřebujeme, nemá pro nás žádnou informační hodnotu a ani v programu nebude potřebná (pro tuto metodu). Ovšem pokud by bylo možné z koncového stavu dále pokračovat, musel by pro tento stav existovat řádek v tabulce.

Aby se jednoduše vytvářela reprezentace této tabulky v programu, očíslujeme také sloupce, místo písmen budeme používat čísla $1, 2, \ldots, 8$.

Vytvoříme si konstanty pro chybový a koncové stavy, aby byl následující kód přehlednější. Tabulku budeme v paměti reprezentovat formou 2D pole, prvky budou typu `int`. Protože indexy pole mají být v jazyce C celočíselné, můžeme buď použít ASCII hodnoty (ale to by znamenalo hodně „širokou“ tabulku), nebo si naprogramujeme konverzní funkci, která jednotlivým znakům přiřadí číslo, v našem případě z

	1	2	3	4	5	6	7	8
	i	f	t	h	e	n	l	s
⇝ 0	1		2		5			
1		10						
2				3				
3	8				4			
4						11		
5							6	
6								7
7					12			
8								13
CH 9								
←10								
←11								
←12								
←13								

Tabulka 2.2: Tabulka přechodů konečného automatu

intervalu 0–7. Obě možnosti jsou použitelné, záleží na jazyce. Případně můžeme použít omezený interval z ASCII, podle znaků povolených v názvech proměnných a jiných identifikátorů.

```
const int
  k_chyba = 9,      // chybový stav a koncové stavy:
  k_if    = 10,
  k_then  = 11,
  k_else  = 12,
  k_this  = 13,
  pocetStavu =  9,  // stavy 0..8
  pocetZnaku =  8;  // znaky 0..7

int tab[pocetStavu][pocetZnaku]; // 2D pole pro tabulku přechodů
int dejZnakCislo();              // funkce vrací čísla podle znaků v záhlaví tabulky
```

Nejdřív je třeba naplnit tabulku přechodů, tedy naše 2D pole. Ovšem to provedeme pouze jednou při spuštění překladače, nikoliv při každém rozpoznávání dalšího klíčového slova. Prože se jedná o řídkou tabulku, bude vhodnější nejdřív celou tabulku naplnit hodnotou `k_chyba`, a pak upravíme ty buňky, ve kterých má být jiná hodnota.

```
void nactiTabulkuPrechodu() {
  for(int i=0; i<pocetStavu; i++)
    for(int j=0; j<pocetZnaku; j++)
      tab[i][j] = k_chyba;

  tab[0][0]=1;          tab[0][2]=2;          tab[0][4]=5;
  tab[1][1]=k_if;
  tab[2][3]=3;
  tab[3][0]=8;          tab[3][4]=4;
  tab[4][5]=k_then;
  tab[5][6]=6;
  tab[6][7]=7;
  tab[7][4]=k_else;
  tab[8][7]=k_this;
}
```

Následující funkce již provádí přesně to, co potřebujeme, tedy rozlišení jednotlivých klíčových slov, přičemž to, co „zbyde“, není klíčové slovo, ale například proměnná (nebo název uživatelského datového typu,...):

```
int lex_klic_slova(string slovo) {
  int stav=0, delka=slovo.length(), poz=0;
  while ((poz<delka)  && (stav != k_chyba)) {
    stav = tab[stav][slovo[poz]];
    poz++;
  }
  switch (stav) {
    case k_if:   symbol.typ = S_IF;   break;
    case k_then: symbol.typ = S_THEN; break;
    case k_else: symbol.typ = S_ELSE; break;
    case k_this: symbol.typ = S_THIS; break;
    default:     symbol.typ = S_ID;              // chyba, tedy přesněji proměnná
  }
  return symbol.typ;
}
```

Poznámka

Všimněte si, že v úvahu bereme i případ, kdy některé klíčové slovo je „prodloužením“ jiného (třebaže v naší množině klíčových slov zrovna takový případ není) – jde to díky podmínce ve smyčce `while`. Například kličová slova `find` a `findstr` by touto metodou bylo také možné správně rozpoznat.

Pokud by funkce nesloužila jen pro rozpoznávání klíčových slov, ale byla by hlavní funkcí lexikální analýzy (třebaže by to bylo nepraktické), a tedy by pracovala přímo se vstupním řetězcem, pak by přicházelo v úvahu pozměnit podmínku:

```
  while ((vstup.znak !='\0')  && (stav < k_chyba)) { ... }
```

Znamená to, že se zastavíme buď v případě, že jsme se dostali na konec řádku, nebo v případě chyby, nebo v některém koncovém stavu (protože všechny koncové stavy mají číslo vyšší než `k_chyba`). Pak by ale bylo třeba otestovat, jestli přece jen při ukončení v koncovém stavu nenásleduje další znak povolený pro proměnné. Například pokud bychom našli řetězec `find`, měli bychom ověřit, jestli za „`d`“ nenásleduje písmeno či číslice, protože se může jednat třeba o proměnnou `findsomething`.

Nebo další varianta: určitě by dávalo smysl použít číslo 0 pro chybový stav. Pak by „prázdné buňky“ byly reprezentovány nulami, což by bylo přirozenější.

Tato metoda je vhodná pro konečné jazyky, například pro odlišení klíčových slov od ostatních identifikátorů. Její velkou výhodou je univerzálnost, tedy snadná rozšiřitelnost jazyka, pro který je vytvořena. V případě, že chceme rozšířit množinu klíčových slov, rozšíříme matici o další řádky a případně sloupce. V programu provádíme změny pouze na datech, nemusíme měnit přímo kód programu (tabulka přechodů může být definovaná v externí knihovně, příp. v textovém či binárním souboru, případná aktualizace by zahrnovala pouze výměnu nebo úpravu tohoto souboru).

Nevýhodou metody je zbytečně velké místo zabrané tabulkou přechodů, většinu místa zabírají buňky představující „chybový“ stav. To se dá řešit implementací tabulky pomocí řídké matice, což však trochu zpomalí překlad.

C) Stav reprezentován proměnnou

Opět se jedná o metodu vhodnou spíše pro konečné jazyky, i když je použitelná i pro jazyky nekonečné. Dá se považovat za modifikaci metody uvedené v předchozím odstavci: v následujícím kódu se některé části budou opakovat, rozdíl bude pouze uvnitř cyklu `while`.

Tabulku přechodů neukládáme do matice, pouze v proměnné zachycujeme stav (může to být celé číslo nebo písmeno, záleží, jaký typ pro proměnnou zvolíme). Roli matice přebírá `switch`. Odpadá nutnost mít v kódu matici s tabulkou, ale zato se poněkud rozšíří cyklus zpracovávající znaky ze vstupu.

Příklad

Automat z předchozího příkladu přepíšeme takto:

```
const int
  k_chyba = 9,       // chybový stav a koncové stavy:
  k_if    = 10,
  k_then  = 11,
  k_else  = 12,
  k_this  = 13,
  pocetStavu =  9,  // stavy 0..8
  pocetZnaku =  8;  // znaky 0..7

int lex_klic_slova(string slovo) {
  int stav=0, delka=slovo.length(), poz=0;
  while ((poz<delka) && (stav != k_chyba)) {
    switch (stav) {
    case 0: switch (znak) {
      case 'I': stav = 1; break;
      case 'T': stav = 2; break;
      case 'E': stav = 5; break;
      default: stav = k_chyba;
    }
    case 1: if (znak=='F') stav = k_if; else stav = k_chyba; break;
    case 2: if (znak=='H') stav = 3;    else stav = k_chyba; break;
    case 3: switch(znak) {
      case 'I': stav = 8; break;
      case 'E': stav = 4; break;
      default: stav = k_chyba;
    }
    case 4: if (znak=='N') stav = k_then;  else stav = k_chyba; break;
    case 5: if (znak=='L') stav = 6;       else stav = k_chyba; break;
    case 6: if (znak=='S') stav = 7;       else stav = k_chyba; break;
    case 7: if (znak=='E') stav = k_else;  else stav = k_chyba; break;
    case 8: if (znak=='S') stav = k_this;  else stav = k_chyba; break;
    default: stav = k_chyba;
    }   // switch(stav)
    poz++;
  }     // while

  switch (stav) {
    case k_if:   symbol.typ = S_IF;   break;
    case k_then: symbol.typ = S_THEN; break;
    case k_else: symbol.typ = S_ELSE; break;
    case k_this: symbol.typ = S_THIS; break;
    default:     symbol.typ = S_ID; // chyba, tedy přesněji proměnná
  }
  return symbol.typ;
}
```

Oproti předchozí metodě je zde výhodou kompaktnější reprezentace tabulky přechodů (nepotřebujeme v paměti místo na celou matici, použijeme jen ty části tabulky, které opravdu potřebujeme), nevýhodou je menší univerzálnost (při změně jazyka musíme zasahovat do kódu, ne jen do dat).

2.3.3 Uplatnění metod na zvolený jazyk

Při výběru mezi metodami výše popsanými se řídíme především podle typu symbolů, které jazyk obsahuje.

Výhodná bývá často kombinace těchto metod – nejdřív použijeme metodu přímého stavového programování (A), a pokud je načtený symbol identifikátor, použijeme některou z metod pro konečné jazyky pro zjištění, zda se jedná o klíčové slovo.

Budeme dále pokračovat v příkladu z kapitoly 2.3.1. Sestavíme funkci `Lex`, jejímž úkolem bude načíst řetězec symbolu a určit jeho typ (identifikovat). V každém koncovém stavu symbolu buď přímo stanovíme hodnotu proměnné `symbol.atrib` deklarované v kapitole 2.3.1, nebo v případě identifikátoru budeme volat funkci `ZpracujID`, která načtený atribut dále zpracuje a určí, zda nejde o klíčové slovo. Na konci každého symbolu se funkce zastaví a ve vyhodnocení vstupu pokračuje, až když je znovu volána.

```
TVstup vstup;        // znak načtený ze souboru
TSymbol symbol;      // zde ukládáme načtený symbol
...

// funkce načte jeden symbol do globální proměnné symbol:
void lex() {
  // funkce dejZnak() byla už volána, v proměnné vstup máme přednačtený znak

  symbol.atrib = "";   // příprava, pokud budeme potřebovat sémantickou informaci

  while (!vstup.konec && isspace(vstup.znak))           // přeskočíme prázdné znaky
    dejZnak();

  if (vstup.znak >= 'A' && vstup.znak <= 'Z') {         // začíná písmenem?
    do {
      symbol.atrib += vstup.znak;
      dejZnak();
    } while ((vstup.znak >= 'A' && vstup.znak <= 'Z')   // příp. přidejte podtržítko
      || (vstup.znak >= '0' && vstup.znak <= '9'));     // resp. isalpha(), isdigit()

    symbol.typ = S_ID;
    lex_klic_slova(symbol.atrib);
  }

  else if (vstup.znak >= '0' && vstup.znak <= '9') {    // začíná číslicí?
    do {
      symbol.atrib += vstup.znak;
      dejZnak();
    } while (vstup.znak >= '0' && vstup.znak <= '9');
    symbol.typ = S_NUM;
  }

  else switch (vstup.znak) {
    case '<':                           // symbol '<' nebo '<=' nebo '<>'
      dejZnak();
      switch (vstup.znak) {
        '>':
          dejZnak();
          symbol.typ = S_NEQ;           // <>
          break;
```

```
        '=':
          dejZnak();
          symbol.typ = S_LQ;            // <=
          break;
      default: symbol.typ = S_LESS;     // <
    };
    ...                                 // podobně všechny ostatní symboly
    default: ...                        // ošetření chyby
  }
}
```

Dále musíme odlišit klíčová slova od ostatních identifikátorů. Funkci můžeme sestavit více způsoby, ten nejméně optimální by byl postupně porovnávat rozpoznávaný řetězec postupně s jednotlivými klíčovými slovy. Ovšem řetězcové operace nejsou zrovna optimální, znamenalo by to, že vlastně k témuž znaku se vracíme vícekrát. Nejhorší situace by nastala, pokud by zpracovávaný řetězec nebyl žádné klíčové slovo – těch návratů by bylo tolik, že by se uživatel pěkně načekal. *Časová složitost*[1] výpočtu by byla příliš velká.

Napíšeme funkci jako konečný automat podle druhé nebo třetí metody z kapitoly 2.3.2.

Příklad

Sestavíme gramatiku, podle ní tabulku přechodů a program, který bude rozpoznávat tento jazyk:
$L = \{\text{begin, end, const, var, if, then, else, print}\}$

$$
\begin{array}{llll}
S \to bA_1 & S \to eA_5 & S \to cA_7 & S \to vA_{11} \\
A_1 \to eA_2 & A_5 \to nA_6 & A_7 \to oA_8 & A_{11} \to aA_{12} \\
A_2 \to gA_3 & A_6 \to d & A_8 \to nA_9 & A_{12} \to r \\
A_3 \to iA_4 & & A_9 \to sA_{10} & \\
A_4 \to n & & A_{10} \to t &
\end{array}
$$

$$
\begin{array}{llll}
S \to iA_{13} & S \to tA_{14} & A_5 \to lA_{17} & S \to pA_{19} \\
A_{13} \to f & A_{14} \to hA_{15} & A_{17} \to sA_{18} & A_{19} \to rA_{20} \\
 & A_{15} \to eA_{16} & A_{18} \to e & A_{20} \to iA_{21} \\
 & A_{16} \to n & & A_{21} \to nA_{22} \\
 & & & A_{22} \to t
\end{array}
$$

Automat bude mít stavy 0...22 přejaté z gramatiky, dále přidáme tyto stavy:

```
const int
  k_chyba = 23,    k_begin = 25,    k_if    = 27,    k_else  = 29,
  k_const = 24,    k_end   = 26,    k_then  = 28,    k_var   = 30,    k_print = 31;
```

Navrhneme deterministickou tabulku přechodů (je v tabulce 2.3, bez řádků pro chybový a koncové stavy) a přepíšeme do datové struktury.

```
const int PocetZnaku = 17;   // Počet znaků, ze kterých se skládají klíčová slova
int tab[22][PocetZnaku];

void NactiTabulku() {
  for (int i = 0; i<22; i++)
    for (int j = 0; j<PocetZnaku; j++)
      tab[i][j] = k_chybovy;

  tab[ 0][ 0] =  1;   tab[ 0][ 1] =  5;   tab[ 0][ 3] = 13;
```

[1] *Časová složitost* znamená náročnost výpočtu algoritmu z hlediska doby jeho trvání v závislosti na délce vstupu. Vyšší časovou složitost má ten algoritmus, jehož provedení v běžném (nebo nejhorším) případě trvá déle.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	B	E	G	I	N	D	C	O	S	T	V	A	R	F	H	L	P
⇝ 0	1	5		13			7			14	11						19
1		2															
2			3														
3				4													
4					24												
5					6											17	
6						25											
7								8									
8					9												
9									10								
10										26							
11												12					
12													27				
13														28			
14															15		
15		16															
16					29												
17									18								
18		30															
19													20				
20				21													
21					22												
22										31							

Tabulka 2.3: Tabulka přechodů pro klíčová slova zvoleného jazyka

```
  tab[ 0][ 6] =  7;   tab[ 0][ 9] = 14;   tab[ 0][10] = 11;
  tab[ 0][16] = 19;   tab[ 1][ 1] =  2;   tab[ 2][ 2] =  3;
  tab[ 3][ 5] =  4;   tab[ 4][ 4] = 24;   tab[ 5][ 4] =  6;
  tab[ 5][15] = 17;   tab[ 6][ 5] = 25;   tab[ 7][ 7] =  8;
  tab[ 8][ 4] =  9;   tab[ 9][ 8] = 10;   tab[10][ 9] = 26;
  tab[11][11] = 12;   tab[12][12] = 27;   tab[13][13] = 28;
  tab[14][14] = 15;   tab[15][ 1] = 16;   tab[16][ 4] = 29;
  tab[17][ 8] = 18;   tab[18][ 1] = 30;   tab[19][12] = 20;
  tab[20][ 5] = 21;   tab[21][ 4] = 22;   tab[22][ 9] = 31;
end;

// pokud zn nepatří do abecedy klíčových slov, vrátí -1, jinak vrací index znaku:
int DejCisloZnaku(char zn) {
  char Index[] = "BEGINDCOSTVARFHLP";
  for (int i=0; i<PocetZnaku; i++) if (Index[i] == zn) return i;
  return -1;
}

void lex_klic_slova(string slovo) {
int
  stav   = 0,             // aktuální stav automatu
  pozice = 1,             // pozice v testovaném řetězci "slovo"
  delka  = length(slovo), // délka řetězce "slovo"
  znak;                   // číslo znaku podle seznamu znaků klíčových slov
  while (pozice <= delka && stav != k_chyba) {
```

```
    znak = DejCisloZnaku(slovo[pozice]);
    if (znak == -1) stav = chybovy;
    else stav = tab[stav][znak];
    pozice++;
  }
  switch (stav) {
    case k_begin: symbol.typ = S_BEGIN; break;
    case k_end:   symbol.typ = S_END;   break;
    case k_const: symbol.typ = S_CONST; break;
    case k_var:   symbol.typ = S_VAR;   break;
    case k_if:    symbol.typ = S_IF;    break;
    case k_then:  symbol.typ = S_THEN;  break;
    case k_else:  symbol.typ = S_ELSE;  break;
    case k_print: symbol.typ = S_PRINT; break;
    default: symbol.typ = S_ID;
  }
}

void InitLex() {     // Tato funkce je volána pouze jednou za celý překlad
  ...                   // otevření vstupního souboru...
  NactiTabulku();       // načteme tabulku přechodů do proměnné tab
  dejZnak();            // přednačteme první znak souboru
}
```

Časová složitost našeho řešení je obecně mnohem nižší než kdybychom postupně porovnávali s různými klíčovými slovy (každý znak slova je zde zpracováván nejvýše jednou), narůstá však prostorová složitost[2], protože v paměti je uložena celá tabulka přechodů automatu. V dnešní době vyšší prostorová složitost již tolik nevadí, a i kdyby, dá se řešit například použitím technik pro zachycení řídké matice (většina prvků tabulky má tutéž hodnotu). Můžeme samozřejmě postupovat také metodou pro konečné jazyky s nižší prostorovou složitostí, která je ukázaná v sekci 2.3.2 na straně 26.

Úkoly

1. Vytvořte regulární gramatiku jazyka celých nezáporných čísel.
2. Podle gramatiky, kterou jste sestrojili v úkolu 1, vytvořte diagram deterministického konečného automatu.
3. Vytvořte regulární gramatiku jazyka reálných nezáporných čísel, celá a reálná část čísla jsou odděleny desetinnou tečkou, která je nepovinná (pak jde o celé číslo), před tečkou nemusí být žádná číslice, za tečkou musí být alespoň jedna číslice.

   Podle této regulární gramatiky vytvořte diagram deterministického konečného automatu.
4. Sestrojte regulární gramatiku a podle ní *deterministický* konečný automat reprezentovaný tabulkou přechodů pro jazyk $L_1 = \{\text{is, then, this}\}$.

   Automat má rozpoznávat jednotlivá slova jazyka, bude mít pro každé slovo jiný koncový stav. Gramatiku vytvořte tak, aby bylo možné konstruovat automat přímo jako deterministický, bez nutnosti další transformace.
5. Naprogramujte konečný automat z úkolu 4 některou z metod z této kapitoly nebo jejich kombinací (metody jsou popsány v podkapitole 2.3.2 od strany 22, možnost kombinace metod v podkapitole 2.3.3 od strany 27).

[2] Jestliže máme dva algoritmy A_1 a A_2 a řekneme, že A_1 má vyšší *prostorovou složitost*, znamená to, že při výpočtu algoritmu A_1 je pro běžné vstupy použito více paměťového prostoru než při výpočtu algoritmu A_2.

6. Sestrojte regulární gramatiku a podle ní deterministický konečný automat pro tyto jazyky:
   - $L_2 = \{\text{if, then, else, elif, end}\}$ (automat reprezentovaný tabulkou symbolů)
   - $L_3 = \{\text{jdi, stop, doprava, doleva}\}$ (automat reprezentovaný tabulkou symbolů)
   - $L_4 = \{\text{read, write, var}\} \cup \{a, \ldots, z\}^+$ (tři klíčová slova a názvy proměnných obsahující pouze malá písmena, alespoň jedno)
   - $L_5 = \{\text{if, write, <, >, <=, >=, <>}\} \cup \{0, \ldots, 9\}^+$ (dvě klíčová slova, relační operátory, celá čísla)
   - $L_6 = \{\text{line, oval, rect, ,, [,]}\} \cup \{0, \ldots, 9\}^+$ (tři klíčová slova, čárka, hranaté závorky, celá čísla)
   - $L_7 = \{+, -, *, /, :=, (,)\} \cup \{0, \ldots, 9\}^+ \cup (\{a, \ldots, z\} \cdot \{a, \ldots, z, 0, \ldots, 9\}^*)$ (matematické výrazy s běžnými aritmetickými operátory a operátorem přiřazení, závorkami, celými čísly a proměnnými – název proměnné začíná písmenem, pak mohou následovat písmena nebo číslice)
   - $L_8 = L_6 \cup L_7$ (v parametrech příkazů z jazyka L_6 mohou být běžné matematické výrazy včetně použití proměnných, hodnotu proměnných lze určit přiřazovacím příkazem)
7. Vyberte si kterýkoliv z jazyků L_2–L_7 z předchozího úkolu a naprogramujte jeho lexikální analýzu některou z metod uvedených v této kapitole (nebo jejich kombinací).
8. Naprogramujte lexikální analýzu jazyka L_8 z úkolu 6 kombinací metod podle podkapitoly 2.3.3 (strana 27).
9. Upravte kód metody přímého stavového programování použitý na načítání čísel (celý kód začíná na straně 27) tak, aby lexikální analyzátor převáděl načtené číslo z řetězcové na číselnou reprezentaci, a to bez použití funkcí poskytovaných programovacím jazykem, ve kterém pracujete. Pro celé číslo bude v symbolu uložena jak jeho číselná hodnota, tak i řetězcové vyjádření. Symbol ukládejte do proměnné následujícího datového typu:

```
struct TSymbol {
  TTypSybolu typ;         // identifikace symbolu
  int cislo;              // atribut ve formátu celého čísla
  string retezec;         // atribut ve formátu řetězce
};
```

   Nápověda: při načítání číslic ve směru zleva iniciujeme proměnnou pro výsledek hodnotou 0 a pak v cyklu využíváme faktu, že pouhým násobením lze číslo řádově zvýšit (v desítkové soustavě tedy násobíme číslem 10).

Kapitola 3

Syntaktická analýza

Rychlý náhled: Lexikální analyzátor rozložil text zdrojového programu na jednotlivé symboly. Úkolem syntaktického analyzátoru je zjistit, jak tyto symboly patří k sobě, tedy sestavit syntaktickou strukturu programu. Symboly jsou zde jakýmisi slovy, ze kterých je třeba sestavit větu – strukturu programu.

V této kapitole se nejdřív budeme zabývat vytvořením základních struktur pro zpracování syntaktické analýzy pomocí bezkontextových gramatik a zásobníkových automatů. Pak upravíme zásobníkový automat tak, aby se snadněji programoval, a probereme způsob samotného naprogramování.

Klíčová slova: Syntaktická analýza, derivační strom, symbol, analýza metodou shora dolů/zdola nahoru, analýza s návratem, deterministická analýza, množina FIRST, množina FOLLOW, množina BEFORE, množina EFF, LL překlad, silná LL(k) gramatika, transformace, faktorizace pravidel, levá rekurze, eliminace pravidel, silná LR(k) gramatika, překladový automat, rozkladová tabulka, metoda přepisu rozkladové tabulky, metoda rekurzivního sestupu.

Cíle studia: Cílem této kapitoly je naučit se navrhnout syntaktickou strukturu zvoleného jazyka a naprogramovat jeho syntaktickou analýzu.

3.1 Princip syntaktické analýzy

V následující tabulce jsou shrnuty nejdůležitější vlastnosti syntaktické analýzy.

Vstup:	*posloupnsot symbolů*
Výstup:	*syntaktická struktura programu ve formě derivačního stromu*
Syntaktické chyby:	souvisí se syntaktickou strukturou programu, chybná posloupnost symbolů (např. $25 := x$ nebo $IF\ x > 2\ ELSE\ y = 3$) ...

Tabulka 3.1: Vlastnosti syntaktické analýzy

Předpokládáme, že zdrojový soubor (nebo jiná forma původního vstupu) byl již předzpracován lexikálním analyzátorem, tedy vstupem je posloupnost symbolů ve vhodné formě, přičemž veškeré součásti nedůležité pro překladač jsou odfiltrovány (například komentáře).

Co je to ta „vhodná forma“? Záleží na tom, jestli jsou lexikální a syntaktický analyzátor v tomtéž průchodu.

- Jsou v jiných průchodech: potřebujeme od lexikálního analyzátoru posloupnost symbolů, například jako dynamický seznam (příp. vektor apod., podle možností programovacího jazyka, ve kterém překladač vytváříme), v horším případě v souboru.
- Jsou v tomtéž průchodu: pak je efektivnější, když lexikální analyzátor je implementován jako funkce, která při každém zavolání dodá JEDEN (následující) symbol. Pak syntaktický analyzátor pracuje tak, že si vyzvedne od lexikálního analyzátoru symbol, zařadí ho do syntaktické struktury programu, pak si vyzvedne další, opět ho zařadí, atd.

3.2 Derivační strom

Syntaktickou strukturu programu budeme popisovat derivačním stromem. Tento pojem již známe z předmětu *Teorie jazyků a automatů*, pro upřesnění uvádíme definici.

Definice (Derivační strom)

Derivační strom derivace v gramatice G je orientovaný acyklický graf s jediným kořenem, do všech ostatních uzlů vstupuje právě jedna hrana, a dále má tyto vlastnosti:

1. Kořen stromu je ohodnocen startovacím symbolem gramatiky.
2. Koncové uzly stromu (listy) jsou ohodnoceny terminálními symboly nebo prázdným řetězcem (ε), všechny ostatní uzly (tj. vnitřní uzly stromu) jsou ohodnoceny neterminálními symboly.
3. Všechny koncové uzly v jakékoliv fázi konstrukce čtené zleva doprava tvoří větnou formu v gramatice G.
4. Jestliže uzly n_1, n_2, …, n_k jsou bezprostřední následníci uzlu n, jsou ohodnoceny symboly A_1, A_2, …, A_k a uzel n je ohodnocen A, pak v množině pravidel gramatiky existuje pravidlo $A \to A_1 A_2 \ldots A_k$.
5. Derivační strom tvoříme zleva doprava a shora dolů, proto není třeba značit orientaci hran.

Z definice plyne, že derivační strom konstruujeme vždy ke konkrétní derivaci.

Příklad

Máme bezkontextovou gramatiku $G = (\{S\}, \{n, i, +, *\}, P, S)$, množina P obsahuje pravidla $S \to S + S \mid S * S \mid n \mid i$.

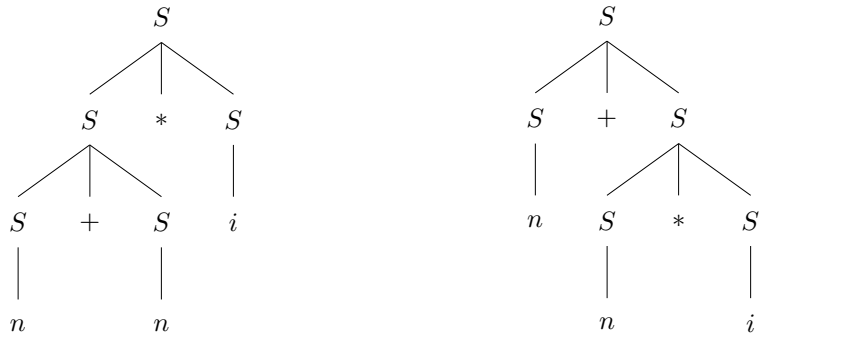
Odvodíme větu $n + n * i$ třemi různými derivacemi a ke každé vytvoříme derivační strom (v derivaci je vždy zvýrazněn ten neterminál, který má být v následujícím kroku přepsán).

Derivace D1: $\mathbf{S} \Rightarrow \mathbf{S} * S \Rightarrow \mathbf{S} + S * S \Rightarrow n + \mathbf{S} * S \Rightarrow n + n * \mathbf{S} \Rightarrow n + n * i$

Derivace D2: $\mathbf{S} \Rightarrow \mathbf{S} * S \Rightarrow S + S * \mathbf{S} \Rightarrow S + \mathbf{S} * i \Rightarrow \mathbf{S} + n * i \Rightarrow n + n * i$

Derivace D3: $\mathbf{S} \Rightarrow S + \mathbf{S} \Rightarrow \mathbf{S} + S * S \Rightarrow n + \mathbf{S} * S \Rightarrow n + n * \mathbf{S} \Rightarrow n + n * i$

Derivační stromy těchto derivací jsou na obrázku 3.1 na straně 34. Jak vidíme, ve všech třech případech je generován stejný výstup (říkáme věta), ale třetí derivace má zcela jiný derivační strom.



(a) Derivační strom pro derivace **D1** a **D2** (b) Derivační strom pro derivaci **D3**

Obrázek 3.1: Derivační stromy pro různé derivace

Obecně platí, že pro tutéž větu (příp. větnou formu) může existovat více derivačních stromů (každý pro jinou derivaci, podle příkladu 3.2 derivace D1, D3), ale také různé derivace mohou mít stejný derivační strom (derivace D1, D2).

Při programování je důležitá jednoznačnost, *determinismus*. Pokud pro dva různé vstupy může existovat více různých možných výstupů (derivačních stromů, syntaktických struktur programu), pak to pro programátora znamená vždy problém.

Definice (Jednoznačná a víceznačná gramatika)

Gramatika je *jednoznačná*, pokud pro každý terminální řetězec, který lze v gramatice vygenerovat (tj. větu), existuje právě jeden derivační strom.

Gramatika je *víceznačná*, pokud není jednoznačná; tj. pokud existuje terminální řetězec patřící do jazyka této gramatiky, ke kterému lze sestrojit více různých derivačních stromů.

Jednoznačné gramatiky se velmi špatně hledají, zvláště pro jazyky, které popisují syntaktickou strukturu programů. Proto se používají víceznačné gramatiky, a pro zajištění jednoznačnosti výstupu překladu máme další možnosti, například stanovení podmínek, za jakých derivace může probíhat. Nejběžnějšími podmínkami jsou výhradní používání levé nebo pravé derivace.

Poznámka

Pro připomenutí: když v každém kroku derivace přepisujeme vždy nejlevější neterminál (ten, který je ve větné formě nejvíce vlevo), používáme *levou derivaci*, když přepisujeme vždy neterminál nejvíce v pravo, používáme *pravou derivaci*.

Pozor – zatím hovoříme o jednoznačnosti výstupu, nikoliv samotného procesu překladu.

Úkol

Je dána gramatika $G = (N, T, P, C)$, kde $N = \{C, V, R\}$,
$T = \{\langle\text{prom}\rangle, \langle\text{num}\rangle, \langle\text{is}\rangle, \langle\text{if}\rangle, \langle\text{then}\rangle, \langle\text{read}\rangle, \langle\text{write}\rangle, \langle\text{plus}\rangle, \langle\text{minus}\rangle, \langle\text{less}\rangle, \langle\text{gt}\rangle, \langle\text{eq}\rangle\}$

pravidla P: $C \to \langle\text{prom}\rangle\,\langle\text{is}\rangle\,V \mid \langle\text{if}\rangle\,V\,R\,V\,\langle\text{then}\rangle\,C \mid \langle\text{read}\rangle\,\langle\text{prom}\rangle \mid \langle\text{write}\rangle\,V$
$V \to V\,\langle\text{plus}\rangle\,V \mid V\,\langle\text{minus}\rangle\,V \mid \langle\text{prom}\rangle \mid \langle\text{num}\rangle$
$R \to \langle\text{less}\rangle \mid \langle\text{gt}\rangle \mid \langle\text{eq}\rangle$

Jedna z možných derivací je například tato:
$C \Rightarrow \langle\text{if}\rangle\, V\, R\, V\, \langle\text{then}\rangle\, C \Rightarrow \langle\text{if}\rangle\, \langle\text{prom}\rangle\, R\, V\, \langle\text{then}\rangle\, C \Rightarrow \langle\text{if}\rangle\, \langle\text{prom}\rangle\, \langle\text{less}\rangle\, V\, \langle\text{then}\rangle\, C \Rightarrow$
$\Rightarrow \langle\text{if}\rangle\, \langle\text{prom}\rangle\, \langle\text{less}\rangle\, \langle\text{num}\rangle\, \langle\text{then}\rangle\, C \Rightarrow \langle\text{if}\rangle\, \langle\text{prom}\rangle\, \langle\text{less}\rangle\, \langle\text{num}\rangle\, \langle\text{then}\rangle\, \langle\text{prom}\rangle\, \langle\text{is}\rangle\, V \Rightarrow$
$\Rightarrow \langle\text{if}\rangle\, \langle\text{prom}\rangle\, \langle\text{less}\rangle\, \langle\text{num}\rangle\, \langle\text{then}\rangle\, \langle\text{prom}\rangle\, \langle\text{is}\rangle\, V\, \langle\text{plus}\rangle\, V \Rightarrow$
$\Rightarrow \langle\text{if}\rangle\, \langle\text{prom}\rangle\, \langle\text{less}\rangle\, \langle\text{num}\rangle\, \langle\text{then}\rangle\, \langle\text{prom}\rangle\, \langle\text{is}\rangle\, \langle\text{num}\rangle\, \langle\text{plus}\rangle\, V \Rightarrow$
$\Rightarrow \langle\text{if}\rangle\, \langle\text{prom}\rangle\, \langle\text{less}\rangle\, \langle\text{num}\rangle\, \langle\text{then}\rangle\, \langle\text{prom}\rangle\, \langle\text{is}\rangle\, \langle\text{num}\rangle\, \langle\text{plus}\rangle\, \langle\text{prom}\rangle$

Jak vidíme, opravdu se v řeší pouze syntaxe. Vygenerovaná věta by mohla po přidání sémantiky odpovídat například příkazu `if X<25 then Y=48+X`

1. Ujasněte si, jaké terminály a neterminály se v gramatice vyskytují. Dokážete obecně říci, jaký jazyk tato gramatika generuje?
2. Projděte si predloženou derivaci. Jde o levou nebo pravou derivaci, nebo žádnou z nich?
3. Vytvořte derivační strom této derivace.
4. Sestrojte pravou derivaci stejné věty a její derivační strom, porovnejte.

3.3 Metody syntaktické analýzy

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

1. Metoda *shora dolů* (Top-Down): derivační strom konstruujeme od kořene k listům, zleva doprava.
2. Metoda *zdola nahoru* (Bottom-Up): postupujeme od listů ke kořeni, avšak také zleva doprava.

Obě metody si zde krátce popíšeme, podrobně se jim budeme věnovat v následujících sekcích této kapitoly a také v dalších kapitolách.

Příklad

Princip obou metod si v následujících příkladech této sekce ukážeme na slově `aabbcc` odvozeném v gramatice $G = (\{S, A, B\},\ \{a, b, c\},\ P,\ S)$ s pravidly

$S \to AB$ ①
$A \to aAb \mid ab$ ②,③
$B \to cB \mid c$ ④,⑤

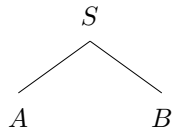
Čísla v kroužcích jsou čísla jednotlivých pravidel. Budou nám sloužit k snadnějšímu odkazování na jednotlivá pravidla.

3.3.1 Metoda shora dolů

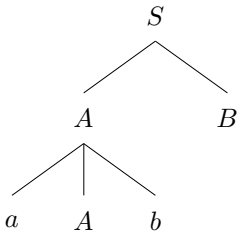
Při použití metody shora dolů (Top-Down) konstruujeme derivační strom věty shora od kořene (ohodnoceného startovacím symbolem gramatiky) dolů k listům, zleva doprava, podle levé derivace:

$S \Rightarrow AB \Rightarrow aAbB \Rightarrow aabbB \Rightarrow aabbcB \Rightarrow aabbcc$

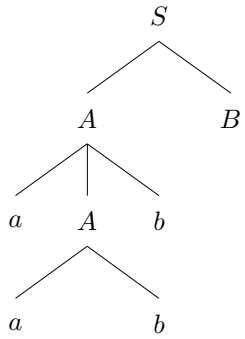
Výstupem syntaktického analyzátoru je derivační strom. Abychom nemuseli mít v paměti uložený celý tento strom (např. ve formě dynamického stromu), použijeme „úspornější“ reprezentaci – posloupnost čísel pravidel, která jsme použili při vytváření derivačního stromu. Protože u metody shora dolů



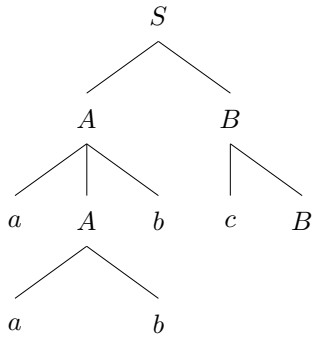
(a) Derivační strom po prvním kroku derivace



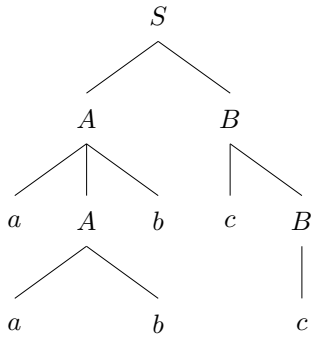
(b) Derivační strom po druhém kroku



(c) Třetí krok



(d) Čtvrtý krok



(e) Pátý krok

Obrázek 3.2: Postup vytvoření derivačního stromu pro levou derivaci

používáme vždy levou derivaci, pak tato posloupnost jednoznačně určuje jak samotnou derivaci, tak i celý derivační strom.

Definice (Lineární rozklad, levý rozklad)

Lineární rozklad věty v gramatice G je některá posloupnost čísel pravidel použitých v derivaci věty v gramatice G.

Levý rozklad věty v gramatice G je posloupnost čísel pravidel použitých v levé derivaci této věty v gramatice G.

Podle příkladu 3.3 bude levý rozklad slova *aabbcc* posloupnost ①, ②, ③, ④, ⑤.

Účelem syntaktické analýzy věty je nalezení derivačního stromu dané věty. Derivační strom nese informaci o tom, jak bychom větu dostali levou či pravou derivací z počátečního symbolu, jak je sestrojena, tedy jaká je její syntaktická struktura.

Definice (Syntaktická analýza metodou shora dolů)

Syntaktická analýza metodou shora dolů je proces nalezení levého rozkladu dané věty.

Zatím jsme pracovali s bezkontextovou gramatikou, ale jak víme, gramatika generuje výstup a nemá žádný vstup. Jenže my potřebujeme, aby překladač zpracovával vstup od uživatele, podle gramatiky zkontroloval, zda tento vstup je správně, a vygeneroval výstup určující syntaktickou strukturu programu. To nezvládne gramatika, potřebujeme automat (ekvivalentní k dané gramatice).

Automat bude jednoduše fungovat tak, že si spustí simulaci derivace v ekvivalentní gramatice, a pokud se mu podaří dojít k témuž výstupu, jaký má na vstupu, ověří tím syntaktickou správnost. Navíc během simulace si konstruuje syntaktickou strukturu daného programu (derivační strom).

Pokud jen generujeme větu v gramatice, můžeme v případě více pravidel se stejnou levou stranou náhodně vybírat. Automat, který provádí simulaci derivace, potřebuje fungovat jednoznačně, žádná náhoda nepřichází v úvahu. Jak tedy dosáhnout toho, abychom v simulaci volili „ta správná“ pravidla?

Problém může nastat v případě, že máme více pravidel se stejnou levou stranou:
$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
Tato pravidla mohou být navzájem různá, nebo třeba mohou začínat stejným podřetězcem. Například podle příkladu 3.3 bychom takto váhali při zpracování větné formy `AB`. Naším úkolem (úkolem překladače) je z těchto pravidel vybrat to správné.

Existují dvě možnosti zajištění determinismu u analýzy metodou shora dolů:

1. *Analýza s návratem* – postupně zkoušíme vhodná pravidla. Nejdřív první, pokračujeme dále ve výpočtu, a když se ukáže, že pravidlo nevyhovuje (dostaneme se do „slepé uličky“), vrátíme se zpátky a vyzkoušíme druhé pravidlo, když to nevyhovuje, tak třetí, ... Tato metoda typu pokus-omyl je sice účinná, ale pomalá, proto se nepoužívá.
2. *Deterministická analýza* – při výběru pravidla se řídíme dalšími informacemi. Může to být „pohled do budoucnosti“, kdy se díváme dále do vstupní posloupnosti symbolů a řídíme se tím, co později dostaneme na vstupu[1], nebo například kontrola obsahu zásobníku (nestačí nám pouze vidět ten symbol, který ze zásobníku vyjímáme, ale i další, které jsou pod ním).

U metody analýzy shora dolů budeme dále vždy používat deterministickou analýzu.

3.3.2 Metoda zdola nahoru

Při použití metody zdola nahoru (Bottom-Up) konstruujeme derivační strom zdola od listů nahoru ke kořeni, také postupujeme zleva doprava, protože tímto směrem se obvykle čte text nebo třeba soubor. Podle příkladu na straně 35 použijeme pravou derivaci:
$S \Rightarrow AB \Rightarrow AcB \Rightarrow Acc \Rightarrow aAbcc \Rightarrow aabbcc$

Stejně jako u první metody, i zde budeme používat lineární rozklad, tentokrát pro pravou derivaci:

Definice (Pravý rozklad)

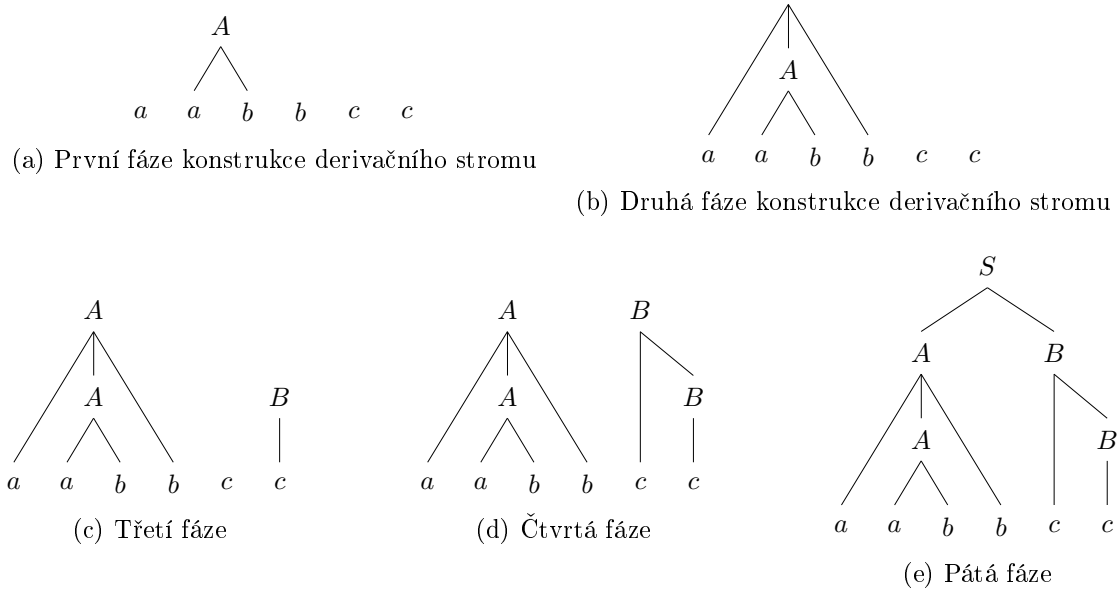
Pravý rozklad věty v gramatice G je **obrácená** posloupnost čísel pravidel použitých v pravé derivaci této věty v gramatice G.

Podle příkladu 3.3 bude pravý rozklad věty *aabbcc* posloupnost čísel ③, ②, ⑤, ④, ① (v pravé derivaci jsme použili pravidla ①, ④, ⑤, ②, ③). Postup vytvoření derivačního stromu touto metodou je naznačen na obrázku 3.3.

Poznámka

Proč „obrácená posloupnost“? Při pravé derivaci v gramatice generujeme větu zprava doleva (přepisujeme vždy neterminál nejvíce vpravo), ale automat čte vstup zleva doprava, tedy tento postup obrací. Proto vytváří obrácenou posloupnost pravidel k té, kterou bychom použili při generování věty.

[1] Opět podle příkladu při derivování větné formy `AB` vidíme na vstupu, že za symbolem `'a'`, na který by zrovna ukazovala čtecí hlava automatu, následuje symbol `'a'`, tedy pro přepis `A` použijeme pravidlo ② a nikoliv ③.



(a) První fáze konstrukce derivačního stromu

(b) Druhá fáze konstrukce derivačního stromu

(c) Třetí fáze

(d) Čtvrtá fáze

(e) Pátá fáze

Obrázek 3.3: Postup vytvoření derivačního stromu pro pravou derivaci

Zbývá definovat syntaktickou analýzu podle dané metody:

Definice (Syntaktická analýza metodou zdola nahoru)
Syntaktická analýza metodou zdola nahoru je proces nalezení pravého rozkladu dané věty.

Podobně jako u metody shora dolů, i zde se postupně dostaneme k automatu a v něm při simulaci musíme rozhodovat mezi pravidly, která chceme použít. Tentokrát však jdeme v derivaci „proti proudu“ (přesněji proti směru šipek), a proto při rozhodování nejde o pravidla se stejnou levou stranou (pro stejný neterminál), ale rozhodujeme se mezi pravidly, která mají podobnou pravou stranu a jsou proto použitelná pro tentýž podřetězec větné formy.

Řešení nutnosti rozhodování je podobné jako u předchozí metody:

1. *Analýza s návratem* – vybereme ve větné formě jeden podřetězec (jako první vybíráme ten, který začíná nejvíc vlevo, je co nejdelší a je shodný s pravou stranou některého pravidla), přepíšeme neterminálem z pravé strany pravidla a pokračujeme v konstrukci derivačního stromu. Když zjistíme, že tento krok nevede k úspěchu, vyzkoušíme jiný podřetězec, ... Tato metoda je jako v předchozím případě také časově náročná, proto ji nebudeme používat.
2. *Deterministická analýza* – využíváme další informace získané při překladu, například obsah nepřečtené části vstupní pásky nebo obsah zásobníku.

Stejně jako u analýzy metodou shora dolů, i v tomto případě budeme volit deterministickou analýzu.

Úkoly

1. Vraťte se ke gramatice v úkolu na straně 34. Očíslujte pravidla tak, jak jsme si ukazovali v této sekci. Pro uvedenou derivaci napište lineární rozklad (pokud jde o levou derivaci, pak levý rozklad), pro vytvořenou pravou derivaci napište pravý rozklad.

2. Podle následující gramatiky vygenerujte jakékoliv slovo dlouhé alespoň 5 znaků levou derivací, pak totéž slovo pravou derivací. K oběma derivacím sestrojte grafy derivačních stromů (pro levou derivaci metodou shora dolů, pro pravou derivaci metodou zdola nahoru) a vypište levý a pravý rozklad.

   $G = (\{S, A, B\},\ \{a, b, c\},\ P,\ S)$
   $S \to aABc \mid bB$
   $A \to aAA \mid aAbA \mid \varepsilon$
   $B \to bB \mid c$

3.4 Pomocné množiny pro syntaktickou analýzu metodou Top-Down

Protože v obou základních metodách syntaktické analýzy budeme výhradně používat deterministickou analýzu, potřebujeme mechanismus, který nám zajistí determinismus při rozhodování mezi pravidly. V této sekci si ukážeme pomocné množiny, na kterých tento mechanismus postavíme.

3.4.1 Množiny FIRST a FOLLOW

Pro analýzu vlastností gramatiky jazyka a také pro konstrukci automatu budeme používat množiny FIRST („první“) a FOLLOW („následník, následující“).

Definice (Množiny FIRST)
Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. Označme α libovolný řetězec nad abecedou $N \cup T$. Potom FIRST(α) je množina terminálních symbolů, jimiž začínají řetězce derivované z α.

Pokud existuje derivace $\alpha \Rightarrow^* \varepsilon$, pak $\varepsilon \in$ FIRST(α).

Množinu FIRST můžeme vytvořit pro jakýkoliv řetězec složený z terminálů a neterminálů dané gramatiky, včetně prázdného řetězce. Poslední část definice řeší právě situaci, kdy zjišťujeme množinu FIRST pro prázdný řetězec nebo lze α na prázdný řetězec přepsat.

Podle definice to vypadá, že bychom pro daný řetězec měli zjistit, co vše se z něj dá derivovat, a potom posbírat terminální symboly na začátcích takto získaných řetězců. Ovšem tento postup by byl nepraktický, takových řetězců může být nekonečně mnoho. Hodí se nám tento deterministický postup:

Označme N_1 množinu všech neterminálních symbolů, pro které existuje ε-pravidlo (nejen přímo $A \to \varepsilon$, ale také *skryté* ε-pravidlo, které umožní neterminál zpracovat na ε až po několika krocích). Pracujeme v gramatice $G = (N, T, P, S)$, postup je iterativní:

$$\text{FIRST}(\varepsilon) = \{\varepsilon\} \tag{3.1}$$

$$\text{FIRST}(a\beta) = \{a\},\ a \in T \tag{3.2}$$

$$\text{FIRST}(A\beta) = \bigcup_{i=1..n} \text{FIRST}(\alpha_i) \quad \text{pro } A \notin N_1, A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n, \tag{3.3}$$

$$\text{FIRST}(A\beta) = \left(\bigcup_{i=1..n} \text{FIRST}(\alpha_i) - \{\varepsilon\} \right) \cup \text{FIRST}(\beta) \quad \text{pro } A \in N_1, A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n, \tag{3.4}$$

První dva řádky postupu zastavují rekurzi, další dva řádky rekurzivně určují, co dělat, když řetězec začíná neterminálem.

Pokud řetězec α začíná terminálem, pak ať už z celého řetězce derivujeme cokoliv, vždy to začíná tímto terminálem. Jestliže však řetězec začíná neterminálem, přenášíme zpracování rekurzívně po derivačním stromě dolů (známá metoda „rozděl a panuj“), tedy zjišťujeme, jakou větu lze dostat, když tento neterminál přepíšeme postupně všemi jeho pravidly. Nesmíme zapomenout ani na ε-pravidlo, po jehož použití neterminál „zmizí“ a ke slovu se dostane zbývající část řetězce (β ve vzorci (3.4)).

Příklad

Je dána gramatika $G = (N, T, P, S)$ s těmito pravidly:

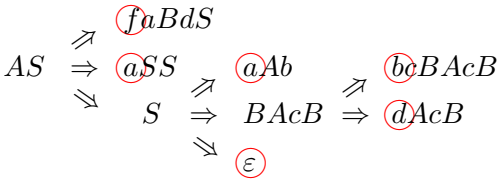
$S \rightarrow aAb \mid BAcB \mid \varepsilon$

$A \rightarrow faBd \mid aS \mid \varepsilon$

$B \rightarrow bcB \mid d$

FIRST(aAb)	$=$	$\{a\}$	podle vzorce (3.2)
FIRST$(BAcB)$	$=$	$\{b, d\}$	podle vzorce (3.3), (3.2)
FIRST(AcB)	$=$	$\{f, a, c\}$	podle vzorce (3.4), (3.2)
FIRST(S)	$=$	$\{a, b, d, \varepsilon\}$	podle vzorce (3.4), (3.2), (3.3), (3.1)
FIRST(AS)	$=$	$\{f, a, b, d, \varepsilon\}$	podle vzorce (3.4), (3.2), (3.3), (3.1)

U množin FIRST v případě komplikovaných gramatik hrozí, že se v možných odvozeních ztratíme a některé symboly se do množiny nedostanou. Pak se hodí někde bokem si vytvořit pomocný strom možných (levých) derivací z daného řetězce. Začátky všech možných derivací pro řetězec AS vidíme na obrázku 3.4.



Obrázek 3.4: Pomocný strom derivací pro výpočet FIRST

Pro pořádek je dobré použít spíše levou derivaci. Nejdřív tedy řešíme neterminál A, který je v řetězci jako první. Pro něj existují tři různá pravidla, proto se v první úrovni strom dělí do tří větví. V prvních dvou větvích jsme hned prvním krokem získali na začátku řetězce terminál, tedy nemusíme pokračovat. V třetí větvi je na začátku řetězce neterminál S, pro který opět existují tři různá pravidla, a pak zbývá ke zpracování už jen jedna větná forma začínající neterminálem. Dál už nemusíme pokračovat, stačí „vyzobat“ terminály, které jsou prvními symboly každé takto získané větné formy, a umístit do množiny FIRST (samozřejmě bez duplicit).

Množiny FIRST obvykle zjišťujeme z pravých stran pravidel nebo jejich podřetězců, budeme je také potřebovat při zjišťování množin FOLLOW definovaných dále.

Poznámka

Pokud píšete množiny FIRST na papír, zkracujte toto „dlouhé slovo“ na první písmeno. Takže například:

F$(aAb) = \{a\}$

Úkol

Je dána tato gramatika:
$G = \{S, A, B\},\ \{a, b, c, d\},\ P,\ S)$
$S \to aAbB \mid AB$
$A \to caAA \mid acB \mid \varepsilon$
$B \to bS \mid bdB \mid \varepsilon$

Sestavte tyto množiny (vpravo máte nápovědu):

1. FIRST$(caAA) =$ (bude to jednoprvková množina)
2. FIRST$(AbB) =$ (bude to tříprvková množina)
3. FIRST$(AB) =$ (množina s pěti prvky)
4. FIRST$(S) =$ (množina s pěti prvky)

S množinami FIRST jsme se už seznámili, teď se podíváme na množiny FOLLOW.

Definice (Množiny FOLLOW)
Je dána gramatika $G = (N, T, P, S)$. Označme \$ symbol ukončení vstupního řetězce (obdobu EOF = End of File), mělo by platit $\$ \in T$.

Nechť A je libovolný neterminál gramatiky G. Potom do množiny FOLLOW(A) řadíme právě ty terminální symboly $a \in T$, které se mohou vyskytovat bezprostředně vpravo od A v nějaké větné formě, tedy existuje derivace $S \Rightarrow^* \beta A a \gamma$.

Pokud je v některé derivaci symbol A posledním symbolem větné formy, do množiny FOLLOW(A) řadíme také symbol konce vstupu \$.

Do množiny FIRST řadíme všechny terminály, kterými může začínat některý testovaný řetězec, do množiny FOLLOW všechny terminály, které v různých derivacích mohou následovat za testovaným neterminálem. Také zde je na konci definice ošetřen případ možného prázdného výsledku – pokud v dané větné formě za testovaným neterminálem nic není (resp. je tam jen symbol konce vstupu).

Zatímco množinu FIRST můžeme určovat u jakéhokoliv řetězce terminálních a neterminálních symbolů (včetně ε), množinu FOLLOW lze určit pouze u neterminálního symbolu, a to vždy *zároveň u všech neterminálů gramatiky*, protože množiny FOLLOW jednotlivých neterminálů jsou vzájemně závislé.

Algoritmus pro výpočet množin FOLLOW je 3-krokový a počítáme vždy tyto množiny pro všechny neterminály gramatiky *najednou*. Předpokládejme gramatiku $G = (N, T, P, S)$, kde $A, B \in N$:

$$\$ \in \text{FOLLOW}(S) \tag{3.5}$$

$$\text{FIRST}(\beta) - \{\varepsilon\} \subseteq \text{FOLLOW}(B), \quad \text{kde } (A \to \alpha B \beta) \in P \tag{3.6}$$

$$\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B), \quad \text{kde } (A \to \alpha B \beta) \in P,\ \beta \Rightarrow^* \varepsilon \tag{3.7}$$

Symbol \$ označuje konec vstupního řetězce, můžeme si ho představit jako konec souboru nebo poslední ukazatel dynamického seznamu ukazující na NULL (v některých programovacích jazycích NIL). Má usnadnit detekci konce vstupu.

Postup popíšeme také slovy:

Krok 1) Do FOLLOW(S), kde S je startovací symbol gramatiky, vložíme symbol konce vstupu \$. Vzhledem k tomu, že derivace vždy začíná symbolem S, je jednoprvkový řetězec S zároveň větnou formou gramatiky a S jako symbol se nachází na jejím konci.

Krok 2) Procházíme postupně všechna pravidla gramatiky, všímáme si pouze pravých stran pravidel a hledáme v nich neterminály, za kterými *něco následuje*: pro každé pravidlo ve tvaru $A \to \alpha B \beta$ umístíme všechny prvky množiny FIRST(β) kromě ε do FOLLOW(B).

Tento krok provádíme postupně pro všechna pravidla gramatiky – hledáme v nich neterminály, za kterými ještě něco následuje (β), a tedy vše terminální, čím může začínat řetězec vytvořený z β, následuje v nějaké větné formě přímo za B.

Jako pomocný nástroj využijeme množiny FIRST. Pozor, pokud v množině FIRST(β) najdeme ε, pak zrovna tento prvek do příslušné množiny FOLLOW nepřeneseme.

Krok 3) Procházíme postupně všechna pravidla gramatiky, teď si pro změnu všímáme všech neterminálů na koncích pravidel, nebo takových, které se na konec pravidla mohou dostat uplatněním ε-pravidel na část pravidla vpravo. Pro každé pravidlo $A \to \alpha B$ nebo $A \to \alpha B \beta$, kde existuje odvození $\beta \Rightarrow^* \varepsilon$, do množiny FOLLOW($B$) zařadíme všechny prvky FOLLOW(A).

Obsah množin FOLLOW vlastně posíláme po derivačním stromě větné formy směrem dolů vždy nejvíce vpravo, pokud je na tom místě neterminál.

První dva kroky provedeme pouze jednou (druhý postupně pro všechna pravidla), třetí krok je nutné provádět rekurzívně tak dlouho, dokud dochází ke změnám v množinách FOLLOW, protože tyto množiny se navzájem ovlivňují. Rekurze samozřejmě skončí po konečném počtu kroků, protože v množinách se nacházejí pouze terminální symboly, kterých je konečně mnoho, a tedy po konečném počtu kroků přestává jejich počet v množinách narůstat.

Příklad

Generování množin FOLLOW ukážeme na gramatice z příkladu na straně 40 s těmito pravidly:

$S \to aAb \mid BAcB \mid \varepsilon$
$A \to faBd \mid aS \mid \varepsilon$
$B \to bcB \mid d$

Podrobný postup:

	1)	2)	3)	4)	5)	6)	7)	8)	
FOLLOW(S) = {	\$,						b, c		}
FOLLOW(A) = {		b,		c					}
FOLLOW(B) = {			f, a, c,		d,	\$,		b	}

1) Podle vzorce (3.5) (S je na konci první větné formy každé derivace). *Krok 1)*
2) Podle vzorce (3.6), pravidlo $S \to a\mathbf{A}b$, kde FIRST(b) = $\{b\}$. *Krok 2)*
3) Podle vzorce (3.6), pravidlo $S \to \mathbf{B}AcB$, kde FIRST(AcB) = $\{f, a, c\}$. –"–
4) Podle vzorce (3.6), pravidlo $S \to B\mathbf{A}cB$, kde FIRST(cB) = $\{c\}$. –"–
5) Podle vzorce (3.6), pravidlo $A \to fa\mathbf{B}d$, kde FIRST(b) = $\{b\}$. –"–
6) Podle vzorce (3.7), pravidlo $S \to BAc\mathbf{B}$ (přenášíme z S do B). *Krok 3)*

7) Podle vzorce (3.7), pravidlo $A \to a\mathbf{S}$ (přenášíme z A do S). –"–
8) Podle vzorce (3.7), pravidlo $S \to BAc\mathbf{B}$ (přenášíme z S do B). –"–

Pro přehlednost uvádíme celé množiny:
FOLLOW$(S) = \{\$, b, c\}$
FOLLOW$(A) = \{b, c\}$
FOLLOW$(B) = \{f, a, c, d, \$, b\}$

Poznámka

Pokud píšete množiny FOLLOW na papír, zkracujte označení na první dvě písmena, například:
FL$(S) = \{\$, b, c\}$

Úkoly

1. Je dána tato gramatika:

   $G = (\{S, A, B\},\ \{a, b, c\},\ P,\ S)$
   $S \to AB$
   $A \to aAb \mid cB \mid ac$
   $B \to cBc \mid AS \mid \varepsilon$

   Nejdřív zjistěte tyto množiny FIRST:

   (a) FIRST$(AB) =$
   (b) FIRST$(AS) =$
   (c) FIRST$(S) =$
   (d) FIRST$(B) =$

   Dále sestrojte množiny FOLLOW pro všechny neterminály gramatiky.

   Nápověda: množiny FOLLOW neterminálů S a B budou dvouprvkové, množina FOLLOW(A) bude obsahovat čtyři prvky.

2. Je dána tato gramatika:

   $G = (\{S, A, B, D, M, P, V\},\ \{b, e, ., ;, c, p, <, >, i, t, w, r, =, +, -, *\},\ R,\ S)$
   $S \to DbAe.$
   $A \to P\,; A \mid \varepsilon$
   $B \to c \mid p$
   $D \to p\,; D \mid \varepsilon$
   $M \to B < B \mid B = B$
   $P \to iMtP \mid wV \mid rp \mid p = V$
   $V \to B + B \mid B - B \mid B * B$

   Upozornění: tečka v prvním pravidle a středník v dalších pravidlech jsou také terminální symboly.

   Podle této gramatiky vytvořte

   (a) FIRST$(bA) =$
   (b) FIRST$(P\,; A) =$
   (c) FIRST$(tP) =$
   (d) FIRST$(< B) =$
   (e) FIRST$(B < B) =$
   (f) FIRST$(S) =$

   A dále sestavte množiny FOLLOW pro všechny neterminály gramatiky. Nezapomeňte, že se vždy vytvářejí všechny množiny FOLLOW zároveň, protože jejich obsah je vzájemně závislý.

3.4.2 Množiny FIRST_k a FOLLOW_k

Pro účely analýzy složitějších gramatik definici množin FIRST a FOLLOW rozšíříme na množiny FIRST_k a FOLLOW_k, index k určuje délku terminálních řetězců řazených do těchto množin. Nadále budeme pro FIRST a FOLLOW (bez uvedeného indexu) předpokládat $k = 1$.

Taktéž platí, že když tyto množiny „čmáráme“ na papír nebo tabuli, zkracujeme na $\mathrm{F}_k(\alpha)$, resp. $\mathrm{FL}_k(A)$.

Definice (Množiny FIRST_k)

Nechť $G = (N, T, P, S)$ je bezkontextová gramatika a $k \geq 1$ přirozené číslo. Označme α libovolný řetězec nad abecedou $N \cup T$. Potom $\mathrm{FIRST}_k(\alpha)$ je množina všech řetězců terminálních symbolů o délce k, jimiž začínají řetězce odvozené z větné formy α.

Pokud lze z α odvodit terminální řetězec o délce menší než k, potom tento řetězec také zařadíme do $\mathrm{FIRST}_k(\alpha)$, tedy jestliže existuje $x \in T^*,\ |x| < k$, kde $\alpha \Rightarrow^* x$, pak $x \in \mathrm{FIRST}_k(\alpha)$.

Množinu $\mathrm{FIRST}_k(\alpha)$ vytváříme podobně jako $\mathrm{FIRST}(\alpha)$, jen místo jednotlivých symbolů pracujeme s řetězci symbolů o délce nejvýše k.

Symbolicky můžeme konstrukci množin FIRST_k zapsat takto:

$$\mathrm{FIRST}_k(\varepsilon) = \{\varepsilon\} \tag{3.8}$$

$$\mathrm{FIRST}_k(w) = \{w\} \text{ pro } w \in T^*,\ |w| \leq k \tag{3.9}$$

$$\mathrm{FIRST}_k(w\beta) = \{w\} \text{ pro } w \in T^*,\ |w| = k,\ \beta \in (N \cup T)^* \tag{3.10}$$

$$\mathrm{FIRST}_k(wA\beta) = \bigcup_{i=1..n} (w \cdot \mathrm{FIRST}_{k-d}(\alpha_i \cdot \beta)) \tag{3.11}$$
$$\text{pro } w \in T^*,\ |w| = d < k,$$
$$A \in N,\ A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

První tři řádky slouží k zastavení rekurze, další řádky jsou rekurzivní. Řádek (3.11) postupu znamená rekurzivní „přenos“ algoritmu na řetězce, na které je neterminál přepisován, používáme zde klasickou metodu „rozděl a panuj“.

Příklad

Gramatika $G = (\{S, A, B\},\ \{a, b, c\},\ P,\ S)$ je definována těmito pravidly:

$S \to ABa$
$A \to ab \mid \varepsilon$
$B \to cB \mid \varepsilon$

Zjistíme postupně množiny FIRST_k pro $k = 1, 2, 3$ řetězců $\varepsilon, a, ab, cB, Ba, AB, ABb, BAc$. Účelem je také to, ať vidíme, jaký je vztah mezi množinami pro různá čísla k téhož řetězce.

Pro $k = 1$ to už umíme:

$\mathrm{FIRST}(\varepsilon) = \{\varepsilon\}$	protože z prázdného slova nic dalšího neodvodíme
$\mathrm{FIRST}(a) = \{a\}$	řetězec začíná terminálem, podle druhého řádku postupu
$\mathrm{FIRST}(ab) = \{a\}$	řetězec začíná terminálem, podle druhého řádku postupu
$\mathrm{FIRST}(cB) = \{c\}$	řetězec začíná terminálem, podle druhého řádku postupu
$\mathrm{FIRST}(Ba) = \{c, a\}$	podle $Ba \Rightarrow cB,\ Ba \Rightarrow a$ (dvě pravidla pro neterminál B)

$\text{FIRST}(AB) = \{a, c, \varepsilon\}$ pokud přepíšeme oba symboly A, B na ε, získáme řetězec ε
$\text{FIRST}(ABb) = \{a, c, b\}$ pokud přepíšeme oba symboly A, B na ε, získáme řetězec b
$\text{FIRST}(BAc) = \{c, a\}$ pokud přepíšeme oba symboly A, B na ε, získáme c, to už tam je

Pro $k = 2$ – nejdřív to jednodušší, kde se nedostaneme do rekurze:

$\text{FIRST}_2(\varepsilon) = \{\varepsilon\}$ beze změny, délka zpracovávaného řetězce je 0
$\text{FIRST}_2(a) = \{a\}$ délka terminálního řetězce je 1, taky podle druhého řádku postupu
$\text{FIRST}_2(ab) = \{ab\}$ délka terminálního řetězce je 2

Následují řetězce, kde budeme muset použít rekurzi. Pro řetězec cB existují dvě možnosti přepisu v jednom kroku:

$cB \Rightarrow ccB \Rightarrow \ldots$ použili jsme pravidlo $B \to cB$
$cB \Rightarrow c$ použili jsme pravidlo $B \to \varepsilon$

Proto bude $\text{FIRST}_2(cB) = \{cc, c\}$. Vůbec nevadí, že druhý prvek je kratší než k, všimněte si druhého odstavce definice.

Podobně pro řetězec Ba jsou pro nás určující tyto první kroky jednotlivých možných derivací:

$Ba \Rightarrow cBa \Rightarrow ccBa$ použili jsme postupně pravidla $B \to cB$, $B \to cB$
$Ba \Rightarrow cBa \Rightarrow ca$ použili jsme postupně pravidla $B \to cB$, $B \to \varepsilon$
$Ba \Rightarrow a$ použili jsme pravidlo $B \to \varepsilon$

$$\begin{array}{lllll} Ba & \Rightarrow & cBa & \Rightarrow & ccBa \\ & \searrow & & \searrow & \\ & & a & & ca \end{array}$$

Proto bude $\text{FIRST}_2(Ba) = \{cc, ca, a\}$. Opět nám nevadí, že poslední prvek je kratší. Vpravo vidíme pomocný strom derivací pro řetězec Ba.

Zbývají tři řetězce, ke každému máme také strom derivací:

$\text{FIRST}_2(AB) = \{ab, cc, c, \varepsilon\}$

$$\begin{array}{lllll} AB & \Rightarrow & abB & & \\ & \searrow & & & \\ & & B & \Rightarrow & cB \quad \Rightarrow \quad ccB \\ & & & \searrow & \quad\quad\;\; \searrow \\ & & & & \varepsilon \quad\quad\;\; c \end{array}$$

$\text{FIRST}_2(ABb) = \{ab, cc, cb, b\}$

$$\begin{array}{lllll} ABb & \Rightarrow & abBb & & \\ & \searrow & & & \\ & & Bb & \Rightarrow & cBb \quad \Rightarrow \quad ccBb \\ & & & \searrow & \quad\quad\;\; \searrow \\ & & & & b \quad\quad\;\; cb \end{array}$$

$\text{FIRST}_2(BAc) = \{cc, ca, ab, c\}$

$$\begin{array}{lllllll} & & & & & \nearrow & ccBAc \\ BAc & \Rightarrow & cBAc & \Rightarrow & cAc & \Rightarrow & cabc \\ & \searrow & & & & & \\ & & Ac & \Rightarrow & abc & & \searrow \quad cc \\ & & & \searrow & & & \\ & & & & c & & \end{array}$$

Množiny FIRST_3 se sestavují podobně. První tři řetězce nepotřebují rekurzi, budou proto jednoprvkové a délka řetězce ve výsledné množině je v našem případě vždy $< k$:

$\text{FIRST}_3(\varepsilon) = \{\varepsilon\}$ $\text{FIRST}_3(a) = \{a\}$ $\text{FIRST}_3(ab) = \{ab\}$

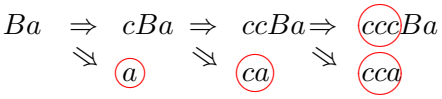
Pro řetězec cB musíme použít jedno z rekurzivních pravidel (nebo ukončit rekurzi), tedy takto:

$cB \Rightarrow ccB \Rightarrow cccB \Rightarrow \ldots$ použili jsme nejméně dvakrát pravidlo $B \to cB$
$cB \Rightarrow ccB \Rightarrow cc$ použili jsme pravidlo $B \to cB$, pak $B \to \varepsilon$
$cB \Rightarrow c$ použili jsme pravidlo $B \to \varepsilon$

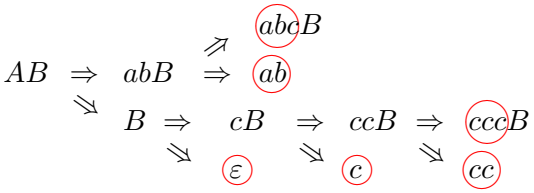
Proto bude $\text{FIRST}_3(cB) = \{ccc, cc, c\}$.

Zbývají čtyři řetězce, ke každému je tu také strom derivací:

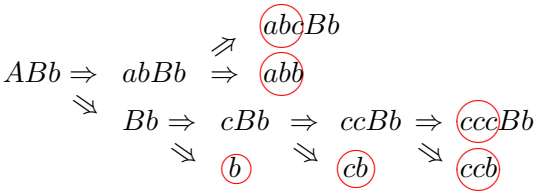
$\text{FIRST}_3(Ba) = \{ccc, cca, ca, a\}$



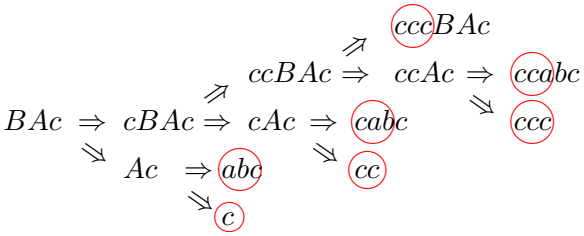
$\text{FIRST}_3(AB) = \{abc, ab, ccc, cc, c, \varepsilon\}$



$\text{FIRST}_3(ABb) = \{abc, abb, ccc, ccb, cb, b\}$



$\text{FIRST}_3(BAc) = \{ccc, cca, cab, cc, abc, c\}$



Teď srovnáme množiny pro tentýž řetězec, ale různá čísla k. Pro řetězce ε, a to nemá moc smysl, podíváme se na ty delší.

$\text{FIRST}(ab) = \{a\}$
$\text{FIRST}_2(ab) = \{ab\}$
$\text{FIRST}_3(ab) = \{ab\}$

$\text{FIRST}(cB) = \{c\}$
$\text{FIRST}_2(cB) = \{cc, c\}$
$\text{FIRST}_3(cB) = \{ccc, cc, c\}$

$\text{FIRST}(AB) = \{a, c, \varepsilon\}$
$\text{FIRST}_2(AB) = \{ab, cc, c, \varepsilon\}$
$\text{FIRST}_3(AB) = \{abc, ab, ccc, cc, c, \varepsilon\}$

$\text{FIRST}(ABb) = \{a, c, b\}$
$\text{FIRST}_2(ABb) = \{ab, cc, cb, b\}$
$\text{FIRST}_3(ABb) = \{abc, abb, ccc, ccb, cb, b\}$

$\text{FIRST}(Ba) = \{c, a\}$
$\text{FIRST}_2(Ba) = \{cc, ca, a\}$
$\text{FIRST}_3(Ba) = \{ccc, cca, ca, a\}$

$\text{FIRST}(BAc) = \{c, a\}$
$\text{FIRST}_2(BAc) = \{cc, ca, ab, c\}$
$\text{FIRST}_3(BAc) = \{ccc, cca, cab, cc, abc, c\}$

Úkol

Projděte si předchozí příkaz a srovnejte obsah množin FIRST_k téhož řetězce pro různá čísla k na konci příkladu. Jaký je mezi nimi vztah? Dá se tento vztah využít pro kontrolu případných zapomenutých prvků v množině pro některé číslo k?

Příklad

Použijeme stejnou gramatiku, na které jsme si dřív ukázali postup vytváření množin FIRST a FOLLOW.

$G = (\{S, A, B\},\ a, b, c, d, f\},\ P,\ S)$
$S \rightarrow aAb \mid BAcB \mid \varepsilon$
$A \rightarrow faBd \mid aS \mid \varepsilon$
$B \rightarrow bcB \mid d$

Naším úkolem je vypsat množiny FIRST_k pro $k \in \{1,2,3\}$ řetězců $aAb, AcB, aS, BAcB$. Pro $k = 1$ je to celkem jednoduché:
$\text{FIRST}(aAb) = \{a\}$
$\text{FIRST}(AcB) = \{f, a, c\}$
$\text{FIRST}(aS) = \{aa, ab, ad, a\}$
$\text{FIRST}(BAcB) = \{b, d\}$

Pro $k = 2$ vycházejí tyto množiny:
$\text{FIRST}_2(aAb) = \{af, aa, ab\}$
$\text{FIRST}_2(AcB) = \{fa, aa, ab, ad, ac, cb, cd\}$
$\text{FIRST}_2(aS) = \{aa, ab, ad, a\}$
$\text{FIRST}_2(BAcB) = \{bc, df, da, dc\}$

Všimněte si, že v žádné z množin pro $k \in \{1,2\}$ není ε (v množině pro $k = 3$ taky nebude). Sice máme pro dva z neterminálů zkracující pravidlo ($S \to \varepsilon$, $A \to \varepsilon$), ale v určených řetězcích je vždy nejméně jeden terminál, takže prázdné slovo se nám do žádné množiny nedostane.

Množiny pro $k = 3$:
$\text{FIRST}_3(aBb) = \{afa, aaa, aab, aad, aab, ab\}$
$\text{FIRST}_3(AcB) = \{fab, fad, aaf, aaa, aab, abc, adf, ada, adc, acb, acd, cbc, cd\}$
$\text{FIRST}_3(aS) = \{aaf, aaa, aab, abc, adf, ada, adc, a\}$
$\text{FIRST}_3(BAcB) = \{bcb, bcd, dfa, daa, dab, dad, dac, dcb, dcd\}$

Mezi množinami FIRST_k pro různá čísla k je poměrně přímočarý vztah a algoritmus jejich výpočtu je velmi podobný. V případě množin FOLLOW_k je algoritmus komplikovanější – jak uvidíme, je přidán jeden nový krok.

Definice (Množiny FOLLOW_k)
Je dána gramatika $G = (N, T, P, S)$ a přirozené číslo $k \geq 1$. Označme \$ symbol ukončení vstupního řetězce, $\$ \in T$, nachází se na konci každé větné formy derivované ze symbolu S.

Nechť A je libovolný neterminál gramatiky G. Potom $\text{FOLLOW}_k(A)$ je množina všech terminálních řetězců α gramatiky G, které se mohou vyskytovat bezprostředně vpravo od A v nějaké větné formě, tedy existuje derivace $S \Rightarrow^* \beta A \alpha \gamma$, $\alpha \in T^k$, $\beta, \gamma \in (N \cup T)^*$.

Pokud lze v G derivovat větnou formu, ve které je délka terminálního řetězce následujícího za A menší než k, potom také tento řetězec řadíme do $\text{FOLLOW}_k(A)$.

Je zřejmé, že všechny terminální řetězce kratší než k, které se dostanou do některé množiny FOLLOW_k, budou zakončeny znakem \$.

Sice budeme také vycházet z postupu pro zjišťování množin FOLLOW, ale musíme přidat ještě čtvrtý krok postupu. Do množin mohou také patřit řetězce kratší než k, pokud jsou „useknuty“ zakončením řetězce derivovaného z α resp. následujícího po A, čtvrtý krok slouží k upřesnění těchto řetězců.

Postup vytvoření množin $\text{FOLLOW}_k(A)$ je následující:

1) Do $\text{FOLLOW}_k(S)$, kde S je startovací symbol gramatiky, vložíme symbol konce vstupu \$.
2) Pro každé pravidlo ve tvaru $B \to \alpha A \beta$ umístíme všechny prvky množiny $\text{FIRST}_k(\beta)$ kromě ε do $\text{FOLLOW}_k(A)$. U všech řetězců kratších než k si poznačíme neterminál, který je pravidlem

přepisován, tj. například pokud v našem případě do množiny $\text{FOLLOW}_k(A)$ řadíme krátký řetězec β, zapíšeme si β^B, protože v pravidle je přepisován neterminál B.

3) Pro každé pravidlo $C \to \alpha A$ nebo $C \to \alpha A\beta$, kde existuje derivace $\beta \Rightarrow^* \varepsilon$, do množiny $\text{FOLLOW}_k(A)$ zařadíme všechny prvky $\text{FOLLOW}_k(C)$.

   Tento krok provádíme rekurzívně tak dlouho, dokud v množinách dochází ke změnám.

4) Nyní vyřešíme „poznámky“ vložené do množin FOLLOW_k v druhém kroku tohoto postupu. Každý opoznámkovaný řetězec β^B vytvořený v předchozích krocích nahradíme množinou řetězců $\text{FIRST}_k(\beta \cdot \text{FOLLOW}_k(B))$, tedy krátké řetězce zřetězíme postupně se všemi prvky množiny FOLLOW_k daného neterminálu z poznámky a zkrátíme na délku k.

   Pokud opět získáme řetězec kratší než k neukončený symbolem \$, přeneseme při prodlužování řetězce také „poznámku“ přidávaných symbolů a tento bod rekurzívně opakujeme až do doby, kdy se ve všech množinách vyskytují pouze řetězce buď dlouhé k znaků, nebo ukončené symbolem \$.

Následuje schematický zápis téhož postupu. Je dána gramatika $G = (N, T, P, S)$ a ve značení předpokládáme $A, B, C \in N$.

$$\$ \in \text{FOLLOW}_k(S) \tag{3.12}$$

$$\text{FIRST}_k(\beta) - \{\varepsilon\} \subseteq \text{FOLLOW}_k(A), \quad \text{kde } B \to \alpha A\beta \tag{3.13}$$

$$\text{FOLLOW}_k(C) \subseteq \text{FOLLOW}_k(A), \quad \text{kde } C \to \alpha A\beta,\ \beta \Rightarrow^* \varepsilon \tag{3.14}$$

$$\text{pokud } \beta^B \in \text{FOLLOW}_k(A), \quad \text{pak } \text{FOLLOW}_k(A) = \text{FOLLOW}_k(A) - \{\beta^B\} \cup \cup\, \text{FIRST}_k(\beta \cdot \text{FOLLOW}_k(B)) \tag{3.15}$$

Třetí a čtvrtý krok se provádějí rekurzívně tak dlouho, dokud probíhají změny.

Příklad

Gramatika $G = (\{S, A, B\},\ \{a, b, c\},\ P,\ S)$ je definována těmito pravidly:

$S \to ABa$

$A \to ab \mid \varepsilon$

$B \to cB \mid \varepsilon$

Množiny FOLLOW pro $k = 1$ už umíme:

$\text{FOLLOW}(S) = \{\$\}$

$\text{FOLLOW}(A) = \{c, a\}$

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

Pro $k = 2$:

Krok 1. Do množiny $\text{FOLLOW}_2(S)$ opět patří řetězec \$. Protože v gramatice se neterminál nevyskytuje na pravé straně žádného pravidla (není rekurzivní), bude to zjevně jediný prvek této množiny: $\text{FOLLOW}_2(S) \ni \$$

Krok 2. Dále budeme postupně procházet všechna pravidla gramatiky a hledat neterminály, „za kterými něco je“.

Z pravidla $S \to ABa$ je zřejmé, že za neterminálem A se nachází řetězec Ba, proto obsah množiny $\text{FIRST}_2(Ba)$ je třeba vložit do $\text{FOLLOW}_2(A)$, a dále za neterminálem B je řetězec a, proto obsah množiny $\text{FIRST}_2(a)$ půjde do $\text{FOLLOW}_2(B)$.

$\text{FIRST}_2(Ba) = \{cc, ca, a\}$, proto $\text{FOLLOW}_2(A) \supseteq \{cc, ca, a^S\}$,
$\text{FIRST}_2(a)$, proto $\text{FOLLOW}_2(B) \supseteq \{a^S\}$.

Všimněte si, že místo rovnítka používáme $\supseteq$ – v daných množinách totiž pravděpodobně bude ještě něco jiného, zatím řešíme první pravidlo gramatiky.

Taktéž si všimněte, že v obou množinách máme řetězec kratší než 2, ten potřebuje „poznámku“. Protože řešíme pravidlo $S \to ABa$, kde je na levé straně neterminál S, pak do poznámky napíšeme právě tento neterminál.

Další dvě pravidla (přepisující neterminál A) můžeme přeskočit, protože na jejich pravých stranách nejsou žádné neterminály. Následuje pravidlo $B \to cB$, kde je na pravé straně netermminál B, ovšem až na konci (za ním se nic nenachází), proto nás v tomto kroku taky nezajímá.

Zatím jsou naše množiny v tomto stavu:
$\text{FOLLOW}_2(S) \supseteq \{\$\}$
$\text{FOLLOW}_2(A) \supseteq \{cc, ca, a^S\}$
$\text{FOLLOW}_2(B) \supseteq \{a^S\}$

Tento krok není rekurzivní, stačilo projít postupně všechna pravidla, v každém zpracovat neterminály na pravých stranách pravidel.

Krok 3. Tento krok probíhá stejně jako pro $k = 1$, prostě kopírujeme obsah množiny neterminálu „před šipkou“ do množiny neterminálu na konci pravé strany pravidla (nebo takového, který se na konec pravé strany může dostat tím, že vše za ním je zlikvidováno použitím ε-pravidel).

V našem případě se na konci pravé strany pravidla vyskytuje pouze symbol B (pravidlo $B \to cB$), ale to by znamenalo kopírovat momentální obsah množiny $\text{FOLLOW}_2(B)$ do sebe samé, což samozřejmě nemá smysl.

Krok 4. V množinách se vyskytují dva řetězce s „poznámkou“, přesněji tentýž řetězec ve dvou různých množinách: a^S. Znamená to, že v obou množinách nahradíme tento řetězec prvky z této množiny:
$\{a^S\} = \text{FIRST}_2(a \cdot \text{FOLLOW}_2(S)) = \{a\$\}$

Do rekurze už jít nemusíme, protože v cílové množině se nevyskytuje žádná další poznámka. Výsledné množiny FOLLOW_2 jsou následující:
$\text{FOLLOW}_2(S) = \{\$\}$
$\text{FOLLOW}_2(A) = \{cc, ca, a\$\}$
$\text{FOLLOW}_2(B) = \{a\$\}$

Vytvoříme několik ukázkových derivací, přičemž na konec každé větné formy přidáme symbol \$ (není to formálně v pořádku, ale chceme zkontrolovat, kde se tento symbol může vyskytovat):
$S \cdot \$ \Rightarrow ABa \cdot \$ \Rightarrow abBa \cdot \$ \Rightarrow abcBa \cdot \$ \Rightarrow abca \cdot \$$
$S \cdot \$ \Rightarrow ABa \cdot \$ \Rightarrow Aa \cdot \$ \Rightarrow aba \cdot \$$
$S \cdot \$ \Rightarrow ABa \cdot \$ \Rightarrow AcBa \cdot \$ \Rightarrow Aca \cdot \$ \Rightarrow abca \cdot \$$
$S \cdot \$ \Rightarrow ABa \cdot \$ \Rightarrow AcBa \cdot \$ \Rightarrow AccBa \cdot \$ \Rightarrow Acca \cdot \$ \Rightarrow abcca \cdot \$$

První derivace je levá, další jsou pravé. Jak vidíme, opravdu se za symbolem S vyskytuje \$, za symbolem A můžeme najít terminální řetězce ca, cc nebo $a\$$, za symbolem B pouze terminální řetězec $a\$$.

Pro $k = 3$:

Krok 1. Opět jednoduché, do množiny pro startovací symbol zařadíme symbol konce vstupu:
$\text{FOLLOW}_3(S) \ni \$$

Krok 2. Budeme postupně procházet všechna pravidla gramatiky a hledat neterminály, „za kterými něco je“. Tato gramatika je opravdu velmi jednoduchá, proto na první pohled vidíme, že nás bude zajímat pouze první pravidlo (v ostatních nemáme na pravých stanách takové neterminály, které nás v tomto kroku zajímají). Takže na pravé straně pravidla $S \to ABa$ jsou dva neterminály:

$A \quad \ldots \quad \mathrm{FIRST}_3(Ba) \subseteq \mathrm{FOLLOW}_3(A), \quad$ tedy $\mathrm{FOLLOW}_3(A) \supseteq \{ccc, cca, ca^S, a^S\}$
$B \quad \ldots \quad \mathrm{FIRST}_3(a) \subseteq \mathrm{FOLLOW}_3(B), \quad$ tedy $\mathrm{FOLLOW}_3(B) \supseteq \{a^S\}$

Množiny FIRST_3 už ovládáme, můžeme si někde „bokem“ vypočíst, že
$\mathrm{FIRST}_3(Ba) = \{ccc, cca, ca, a\}$, což využijeme pro množinu $\mathrm{FOLLOW}_3(A)$,
$\mathrm{FIRST}_3(a) = \{a\}$, což využijeme pro množinu $\mathrm{FOLLOW}_3(B)$.

Krok 3. Stejně jako pro nižší čísla k, i zde v tomto kroku vlastně v této gramatice nemáme co dělat.
Krok 4. Máme zde dva řetězce s poznámkou: ca^S, a^S. Vzorec je tentýž:
$\{ca^S\} = \mathrm{FIRST}_3(ca \cdot \mathrm{FOLLOW}_3(S)) = \{ca\$\}$
$\{a^S\} = \mathrm{FIRST}_3(a \cdot \mathrm{FOLLOW}_3(S)) = \{a\$\}$

Provedeme nahrazení a vypíšeme výsledné množiny:
$\mathrm{FOLLOW}_3(S) = \{\$\}$
$\mathrm{FOLLOW}_3(A) = \{ccc, cca, ca\$, a\$\}$
$\mathrm{FOLLOW}_3(B) = \{a\$\}$

Příklad

Použijeme stejnou gramatiku, na které jsme v příkladech na straně 40, 42 a 46 ukázali postup vytváření množin FIRST, FOLLOW a FIRST_k. Vypočteme množiny FOLLOW_2 a FOLLOW_3.

$G = (\{S, A, B\},\ \{a, b, c, d, f\},\ P,\ S)$
$S \ \to\ aAb \mid BAcB \mid \varepsilon$
$A \ \to\ faBd \mid aS \mid \varepsilon$
$B \ \to\ bcB \mid d$

množiny FOLLOW:
$\mathrm{FOLLOW}(S) = \{\$, b, c\}$
$\mathrm{FOLLOW}(A) = \{b, c\}$
$\mathrm{FOLLOW}(B) = \{f, a, c, d, \$, b\}$

Sestavení množin FOLLOW_2 si opět rozložíme do jednotlivých kroků.

Krok 1. Do množiny pro startovací symbol zařadíme symbol konce vstupu:
$\mathrm{FOLLOW}_2(S) \ni \$$

Krok 2. Projdeme jednotlivá pravidla, hledáme neterminály, za kterými něco následuje. Jde o první dvě pravidla pro neterminál S a první pravidlo pro neterminál A:

- pravidlo $S \to aAb$, na pravé straně je neterminál A, za ním následuje řetězec b:
  $\mathrm{FIRST}_2(b) = \{b\}$, kratší než 2, proto použijeme neterminál na levé straně pravidla jako poznámku:
  $\{b^S\} \subseteq \mathrm{FOLLOW}_2(A)$
- pravidlo $S \to BAcB$, na pravé straně je neterminál B, za ním následuje řetězec AcB:
  $\mathrm{FIRST}_2(AcB) = \{fa, af, aa, ab, ad, ac, cb, cd\}$, proto
  $\{fa, af, aa, ab, ad, ac, cb, cd\} \subseteq \mathrm{FOLLOW}_2(B)$
- pravidlo $S \to BAcB$, na pravé straně je neterminál A, za ním následuje řetězec cB:
  $\mathrm{FIRST}_2(cB) = \{cb, cd\}$, proto $\{cb, cd\} \subseteq \mathrm{FOLLOW}_2(A)$

- pravidlo $A \to faBd$, na pravé straně je neterminál B, za ním následuje řetězec d:
  $\text{FIRST}_2(d) = \{d\}$, kratší než 2, použijeme neterminál na levé straně pravidla jako poznámku:
  $\{d^A\} \subseteq \text{FOLLOW}_2(B)$

Po druhém kroku jsou množiny v tomto stavu:
$\text{FOLLOW}_2(S) \supseteq \{\$\}$
$\text{FOLLOW}_2(A) \supseteq \{b^S, cb, cd\}$
$\text{FOLLOW}_2(B) \supseteq \{fa, af, aa, ab, ad, ac, cb, cd, d^A\}$

Krok 3. Hledáme na pravých stranách pravidel takové neterminály, které buď jsou na konci, nebo se mohou na konec dostat tak, že celý zbytek pravidla by byl přepsán ε-pravidly. V této gramatice jde o druhé pravidlo neterminálu S, druhé pravidlo neterminálu A a první pravidlo neterminálu B.

- pravidlo $S \to BAcB$, na konci pravé strany je neterminál B, tedy celý obsah množiny $\text{FOLLOW}_2(S)$ zkopírujeme do množiny $\text{FOLLOW}_2(B)$:
  $\$ \in \text{FOLLOW}_2(B)$
- pravidlo $A \to aS$, na konci pravé strany je neterminál S, tedy celý obsah množiny $\text{FOLLOW}_2(A)$ zkopírujeme do množiny $\text{FOLLOW}_2(S)$:
  $\{b^S, cb, cd\} \subseteq \text{FOLLOW}_2(S)$
- pravidlo $B \to bcB$ ve skutečnosti nemusíme řešit, protože by znamenalo kopírování obsahu množiny $\text{FOLLOW}_2(B)$ do sebe samé.

 Mnemotechnická pomůcka: ve směru šipky.

Ovšem pozor, tento krok je třeba provádět rekurzívně tak dlouho, dokud v množinách probíhají změny. Takže znovu:

- pravidlo $S \to BAcB$, na konci pravé strany je neterminál B, v množině $\text{FOLLOW}_2(S)$ došlo ke změně (byly přidány nové řetězce z množiny $\text{FOLLOW}_2(A)$), proto přidáváme:
  $\{\$, b^S, cb, cd\} \subseteq \text{FOLLOW}_2(B)$
- pravidlo $A \to aS$, na konci pravé strany je S, ale pro A už ke změnám nedošlo, nemusíme řešit.
- pravidlo $B \to bcB$ opět nemusíme řešit.

Správně bychom měli provést ještě jeden průchod třetího kroku, protože došlo ke změně v množině $\text{FOLLOW}_2(B)$, ale z této množiny se už nic netransportuje, tedy po třetím kroku mají množiny tento obsah:
$\text{FOLLOW}_2(S) \supseteq \{\$, b^S, cb, cd\}$
$\text{FOLLOW}_2(A) \supseteq \{b^S, cb, cd\}$
$\text{FOLLOW}_2(B) \supseteq \{fa, af, aa, ab, ad, ac, cb, cd, d^A, \$, b^S\}$

Krok 4. Vyřešíme poznámky. V množinách se nám vyskytují dvě: b^S, d^A. Tyto řetězce je třeba nahradit následujícími (pod)množinami řetězců:
$\{b^S\} = \text{FIRST}_2(b \cdot \text{FOLLOW}_2(S)) = \text{FIRST}_2(b\$, bb^S, bcb, bcd) = \{b\$, bb, bc\}$
$\{d^A\} = \text{FIRST}_2(d \cdot \text{FOLLOW}_2(A)) = \text{FIRST}_2(db^S, dcb, dcd) = \{db, dc\}$

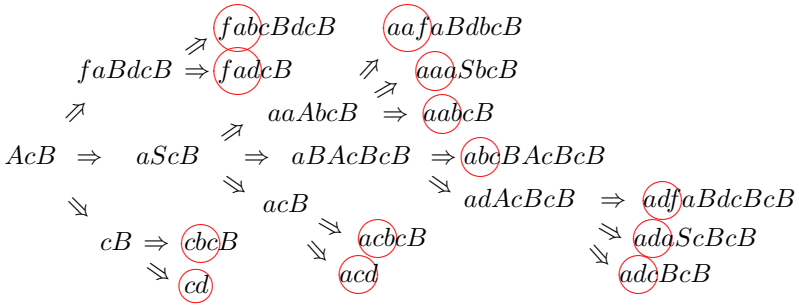
Výsledná podoba množin FOLLOW_2 je následující:
$\text{FOLLOW}_2(S) = \{\$, b\$, bb, bc, cb, cd\}$
$\text{FOLLOW}_2(A) = \{b\$, bb, bc, cb, cd\}$
$\text{FOLLOW}_2(B) = \{fa, af, aa, ab, ad, ac, cb, cd, db, dc, \$, b\$, bb, bc\}$

Zbývá to nejtěžší, množiny FOLLOW_3.

Krok 1. Do množiny pro startovací symbol zařadíme symbol konce vstupu:
$\text{FOLLOW}_3(S) \ni \$$

Krok 2. Projdeme jednotlivá pravidla, hledáme neterminály, za kterými něco následuje, což platí pro tato pravidla:

- $S \to aAb$, neterminál A: $\text{FIRST}_3(b) = \{b\}$, kratší než 3, proto použijeme neterminál na levé straně pravidla jako poznámku:
  $\{b^S\} \subseteq \text{FOLLOW}_3(A)$
- pravidlo $S \to BAcB$, neterminál B: sestrojit množinu $\text{FIRST}_3(AcB)$ není zrovna jednoduché, pomůžeme si stromem možných odvození.
  $\text{FIRST}_3(AcB) = \{fab, fad,$
  $aaf, aaa, aab, abc, adf,$
  $ada, adc, acb, acd, cbc, cd\}$
  Tyto řetězce zařadíme, u jednoho kratšího přidáme poznámku S, protože právě řešíme pravidlo přepisující tento neterminál:



  $\{fab, fad, aaf, aaa, aab, abc, adf, ada, adc, acb, acd, cbc, cd^S\} \subseteq \text{FOLLOW}_3(B)$
- totéž pravidlo, následující neterminál A: $\text{FIRST}_3(cB) = \{cbc, cd\}$, proto
  $\{cbc, cd^S\} \subseteq \text{FOLLOW}_3(A)$
- pravidlo $A \to faBd$, neterminál B: za tímto neterminálem je jednoznakový řetězec d, proto
  $\{d^A\} \subseteq \text{FOLLOW}_3(B)$

Po druhém kroku jsou množiny v tomto stavu:
$\text{FOLLOW}_3(S) \supseteq \{\$\}$
$\text{FOLLOW}_3(A) \supseteq \{b^S, cbc, cd^S\}$
$\text{FOLLOW}_3(B) \supseteq \{fab, fad, aaf, aaa, aab, abc, adf, ada, adc, acb, acd, cbc, cd^S, d^A\}$

Krok 3. Projdeme všechna pravidla, hledáme neterminály, které jsou na konci nebo se na konec mohou dostat uplatněním ε-pravidel.

- $S \to BAcB$, na konci pravidla je B:
  $\$ \in \text{FOLLOW}_3(B)$
- $A \to aS$, na konci pravidla je S:
  $\{b^S, cbc, cd^S\} \subseteq \text{FOLLOW}_3(S)$
- znovu $S \to BAcB$, neterminál B:
  $\{\ldots, b^S, cbc, cd^S\} \subseteq \text{FOLLOW}_3(B)$

Po třetím kroku mají množiny FOLLOW_3 tento obsah:
$\text{FOLLOW}_3(S) \supseteq \{\$, b^S, cbc, cd^S\}$
$\text{FOLLOW}_3(A) \supseteq \{b^S, cbc, cd^S\}$
$\text{FOLLOW}_3(B) \supseteq \{fab, fad, aaf, aaa, aab, abc, adf, ada, adc, acb, acd, cbc, cd^S, d^A, \$, b^S\}$

Krok 4. Zbývá vyřešit poznámky:
$\{b^S\} = \text{FIRST}_3(b \cdot \text{FOLLOW}_3(S)) = \{b\$, bb^S, bcb, bcd\} = \{b\$, bb\$, bbb, bbc, bcb, bcd\}$
$\{cd^S\} = \text{FIRST}_3(cd \cdot \text{FOLLOW}_3(S)) = \{cd\$, cdb, cdc\}$
$\{d^A\} = \text{FIRST}_3(d \cdot \text{FOLLOW}_3(A)) = \{db^S, dcb, dcd\} = \{db\$, dbb, dbc, dcd\}$

Všimněte si, že tentokrát jsme se i v tomto kroku dostali k rekurzi, protože po prvním kroku se nám do množin opět dostaly poznámky. V druhém kroku se už však vyřešily.

Finální tvar množin FOLLOW_3 je následující:
$\text{FOLLOW}_3(S) = \{\$, b\$, bb\$, bbb, bbc, bcb, bcd, cbc, cd\$, cdb, cdc\}$
$\text{FOLLOW}_3(A) = \{b\$, bb\$, bbb, bbc, bcb, bcd, cbc, cd\$, cdb, cdc\}$
$\text{FOLLOW}_3(B) = \{fab, fad, aaf, aaa, aab, abc, adf, ada, adc, acb, acd, cbc, cd\$, cdb, cdc, db\$, dbb, dbc, dcd, \$, b\$, bb\$, bbb, bbc, bcb, bcd\}$
Čím vyšší je číslo k, tím víc prvků mají tyto množiny a stávají se o to nepřehlednější.

Úkoly

1. Prostudujte si předchozí příklad a srovnejte množiny FOLLOW a FOLLOW_2 vždy pro tentýž neterminál. Jakou podobnost vidíte?

   Všimněte si, že vyšší číslo k znamená vlastně upřesnění, ať už se jedná o množiny FIRST nebo FOLLOW. Například v množině $\text{FOLLOW}(A)$ je terminál b, což znamená, že za neterminálem A může v některé větné formě následovat právě tento terminál, ale není zde řečeno, jaký další symbol může být dále. Množina $\text{FOLLOW}_2(A)$ obsahuje terminální řetězce $b\$$, bb a bc, ale neobsahuje řetězce ba, bd ani bf, čehož využijeme při syntaktické kontrole vstupu od uživatele.

2. Je dána tato gramatika:

   $G = (\{S, A, B, C\},\ \{a, b, c, d\},\ P,\ S)$ s pravidly
   $S \to AaaB \mid aS \mid \varepsilon$
   $A \to cA \mid bS \mid \varepsilon$
   $B \to CA \mid Bd$
   $C \to SdB \mid c \mid \varepsilon$

   Zjistěte

   (a) množiny FIRST a FIRST_2 řetězců A, CA, AaB, dB, SdB,
   (b) všechny množiny FOLLOW, a pokud se nudíte, tak i množiny FOLLOW_2.

   Pro kontrolu: díky „kopírovacímu“ kroku by měly být všechny množiny FOLLOW stejné.

3. Je dána tato gramatika:

   $G = (\{S, A, B, C\},\ \{a, b, c, d\},\ P,\ S)$ s pravidly
   $S \to abA \mid cbBC$
   $A \to cA \mid dB \mid a$
   $B \to bBS \mid a$
   $C \to cC \mid cBd \mid \varepsilon$

   Zjistěte

   (a) množiny FIRST a FIRST_2 řetězců C, cA, S, BS, cBd,
   (b) všechny množiny FOLLOW a FOLLOW_2.

3.5 LL(k) překlady

V sekci 3.3 od strany 35 jsme si definovali dvě základní metody syntaktické analýzy – metodu shora dolů (Top-Down) a zdola nahoru (Bottom-Up), podle toho, jak konkrétně je konstruován derivační strom. V této sekci se budeme věnovat první zmíněné metodě. Zkratka LL(k) znamená:

- **L**eft to Right – vstupní text (soubor) čteme zleva doprava,
- **L**eft Parse – vytváříme levý rozklad,
- při rozhodování mezi pravidly potřebujeme vidět nejvýše k znaků z nepřečtené části vstupu.

3.5.1 $LL(k)$ gramatiky

Nejdřív se podíváme na několik definic, abychom si ujasnili, co to vlastně LL(k) překlad je.

Definice ($LL(k)$ **gramatika)**
Bezkontextová gramatika $G = (N, T, P, S)$ je typu LL(k), jestliže ji lze použít pro deterministickou syntaktickou analýzu metodou shora dolů (vytváříme levý rozklad) a v průběhu analýzy při rozhodování mezi pravidly, která lze použít při konstrukci syntaktické struktury vstupu, je nutno znát vždy nejvýše k symbolů ze vstupu.

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

Tato definice je sice vystihující, chybí v ní však náznak konstrukce důkazu, že určitá konkrétní gramatika je $LL(k)$. Podíváme se na jinou definici LL(k) gramatiky.

Definice ($LL(k)$ **gramatika)**
Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. G je $LL(k)$ gramatika pro nějaké celé nezáporné číslo k, jestliže v případě existence dvou levých derivací
$S \Rightarrow^* wA\alpha \Rightarrow w\beta\alpha \Rightarrow^* wx$
$S \Rightarrow^* wA\alpha \Rightarrow w\gamma\alpha \Rightarrow^* wy\ , \qquad$ kde $A \in N,\ w, x, y \in T^*,\ \alpha, \beta, \gamma \in (N \cup T)^*$
takových, že $\mathrm{FIRST}_k(x) = \mathrm{FIRST}_k(y)$, vždy platí $\beta = \gamma$.

Obě množiny FIRST_k terminálních řetězců x a y jsou jednoprvkové (protože x a y jsou terminální řetězce), proto je v definici u vztahu s množinami FIRST_k symbol '='.

Až po větnou formu $wA\alpha$ jsou obě derivace shodné. Když v této větné formě máme pro neterminál A na výběr mezi pravidly $A \to \beta$ a $A \to \gamma$ a vygenerované části slov x a y nedokážeme rozlišit podle prvních k symbolů, pak nesmí být rozlišitelná ani tato pravidla (jinými slovy, u pravidel $A \to \beta$ a $A \to \gamma$ nedokážeme odlišit, kdy které použít, proto musí jít o jedno a totéž pravidlo).

Věta
Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. G je $LL(k)$ gramatika pro nějaké celé nezáporné číslo k právě tehdy, jestliže pro každá dvě různá pravidla se stejnou levou stranou $A \to \beta \mid \gamma$ a každou levou derivaci $S \Rightarrow^* \alpha wA\alpha$ platí

$$\mathrm{FIRST}_k(\beta \cdot \alpha) \cap \mathrm{FIRST}_k(\gamma \cdot \alpha) = \emptyset. \tag{3.16}$$

Důkaz: Podle definice 3.5.1 platí implikace $(\text{FIRST}_k(x) = \text{FIRST}_k(y)) \Longrightarrow (\beta = \gamma)$. Tuto implikaci upravíme podle pravidla $(A \to B) \iff (\neg B \to \neg A)$ do následujícího tvaru a provedeme další ekvivalentní úpravy:

$$\begin{aligned} \neg(\beta = \gamma) &\implies \neg(\text{FIRST}_k(x) = \text{FIRST}_k(y)) \\ \beta \neq \gamma &\implies \text{FIRST}_k(x) \neq \text{FIRST}_k(y) \end{aligned} \tag{3.17}$$

Vztah (3.17) nám říká, že když jsou dvě pravidla navzájem různá, pak pro *všechny* odvozené terminální řetězce řetězce platí, že jsou z hlediska použití těchto dvou pravidel odlišitelné podle svých prvních k symbolů.

Protože víme, že $\beta \cdot \alpha \Rightarrow^* x$ a $\gamma \cdot \alpha \Rightarrow^* y$, vztah (3.17) platí i pro řetězce $\beta \cdot \alpha$ a $\gamma \cdot \alpha$, čímž dostáváme vztah

$$\text{FIRST}_k(\beta \cdot \alpha) \cap \text{FIRST}_k(\gamma \cdot \alpha) = \emptyset.$$

□

Věta na straně 54 již může vypadat jako schéma postupu testování, zda daná gramatika je typu $LL(k)$. Ovšem museli bychom otestovat všechna terminální slova v gramatice odvoditelná, což u nekonečného jazyka není možné. Ve skutečnosti existuje „konečný“ postup testování, ten zde však nebudeme uvádět.

3.5.2 Silné $LL(k)$ gramatiky

Silné $LL(k)$ gramatiky jsou $LL(k)$ a navíc splňují ještě jednu důležitou vlastnost – při jejich zpracování nám pro deterministickou analýzu stačí pouze informace ze vstupu (samozřejmě kromě informací, které nám nabízejí samotná pravidla gramatiky), a to nejvýše k symbolů, nemusíme se řídit dalšími informacemi.

Definice (Silná LL(k) gramatika)

Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. G je silná LL(k) gramatika, jestliže pro jakákoliv dvě pravidla se stejnou levou stranou $A \to \alpha$, $A \to \beta$, kde $\alpha \neq \beta$, platí

$$\text{FIRST}_k(\alpha \cdot \text{FOLLOW}_k(A)) \;\cap\; \text{FIRST}_k(\beta \cdot \text{FOLLOW}_k(A)) \;=\; \emptyset \tag{3.18}$$

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

Postup je následující:

- Pro všechna pravidla $A \to \beta$ sestrojíme porovnávací množiny $\text{FIRST}_k(\beta \cdot \text{FOLLOW}_k(A))$.
- Porovnáme tyto množiny vždy po dvou v rámci skupiny pravidel přepisujících stejný symbol, musí vždy vyjít prázdná množina.

Příklad

Zjistěte, zda je tato gramatika silná LL(k) pro nějaké číslo k.
$G = (\{S, A, B\},\ \{a, b, c, d\},\ P,\ S)$ s pravidly

$S \to abAc \mid BaBA \mid \varepsilon$	$\text{FOLLOW}(S) = \{\$\}$
$A \to aA \mid \varepsilon$	$\text{FOLLOW}(A) = \{c, \$\}$
$B \to aAc \mid d$	$\text{FOLLOW}(B) = \{a, \$\}$

Vytvoříme testovací množiny pro jednotlivá pravidla podle vzorce $\text{FIRST}(\alpha \cdot \text{FOLLOW}(A))$ pro každé pravidlo $A \to \alpha$:

S:	$S \to abAc$	$\text{FIRST}(abAc \cdot \text{FOLLOW}(S)) = \{a\}$
	$S \to BaBA$	$\text{FIRST}(BaBA \cdot \text{FOLLOW}(S) = \{a, d\}$
	$S \to \varepsilon$	$\text{FIRST}(\varepsilon \cdot \text{FOLLOW}(S) = \{\$\}$
A:	$A \to aA$	$\text{FIRST}(aA \cdot \text{FOLLOW}(A) = \{a\}$
	$A \to \varepsilon$	$\text{FIRST}(\varepsilon \cdot \text{FOLLOW}(A) = \{c, \$\}$
B:	$B \to aAc$	$\text{FIRST}(aAc \cdot \text{FOLLOW}(B) = \{a\}$
	$B \to d$	$\text{FIRST}(d \cdot \text{FOLLOW}(B) = \{d\}$

Nejdřív použijeme vzorec pro $k = 1$.

$\text{FIRST}(abAc \cdot \text{FOLLOW}(S)) \cap \text{FIRST}(BaBA \cdot \text{FOLLOW}(S)) = \{a\} \cap \{a, d\} = \{a\}$

Nemusíme pokračovat, už teď je jasné, že gramatika není silná $LL(1)$. Zvolíme proto $k = 2$ a pokračujeme v testování. Množiny FOLLOW_2:

$\text{FOLLOW}_2(S) = \{\$\}$

$\text{FOLLOW}_2(A) = \{c^S, c^B, \$\} = \{c\$, ca, \$\}$

$\text{FOLLOW}_2(B) = \{aa, ad, a^S, \$\} = \{aa, ad, a\$, \$\}$

Vytvoříme testovací množiny jednotlivých pravidel pro $k = 2$:

S:	$S \to abAc$	$\text{FIRST}_2(abAc \cdot \text{FOLLOW}_2(S)) = \{ab\}$
	$S \to BaBA$	$\text{FIRST}_2(BaBA \cdot \text{FOLLOW}_2(S)) = \{aa, ac, da\}$
	$S \to \varepsilon$	$\text{FIRST}_2(\varepsilon \cdot \text{FOLLOW}_2(S) = \{\$\}$
A:	$A \to aA$	$\text{FIRST}_2(aA \cdot \text{FOLLOW}_2(A)) = \{aa, ac, a\$\}$
	$A \to \varepsilon$	$\text{FIRST}_2(\varepsilon \cdot \text{FOLLOW}_2(A)) = \{c\$, ca, \$\}$
B:	$B \to aAc$	$\text{FIRST}_2(aAc \cdot \text{FOLLOW}_2(B)) = \{aa, ac\}$
	$B \to d$	$\text{FIRST}_2(d \cdot \text{FOLLOW}_2(B)) = \{da, d\$\}$

A pustíme se do porovnávání:

$\text{FIRST}_2(abAc \cdot \text{FOLLOW}_2(S)) \cap \text{FIRST}_2(BaBA \cdot \text{FOLLOW}_2(S)) = \{ab\} \cap \{aa, ac, da\} = \emptyset$

$\text{FIRST}_2(abAc \cdot \text{FOLLOW}_2(S)) \cap \text{FIRST}_2(\varepsilon \cdot \text{FOLLOW}_2(S)) = \{ab\} \cap \{\$\} = \emptyset$

$\text{FIRST}_2(BaBA \cdot \text{FOLLOW}_2(S)) \cap \text{FIRST}_2(\varepsilon \cdot \text{FOLLOW}_2(S)) = \{aa, ac, da\} \cap \{\$\} = \emptyset$

$\text{FIRST}_2(aA \cdot \text{FOLLOW}_2(A)) \cap \text{FIRST}_2(\varepsilon \cdot \text{FOLLOW}_2(A)) = \{aa, ac, a\$\} \cap \{c\$, ca, \$\} = \emptyset$

$\text{FIRST}_2(aAc \cdot \text{FOLLOW}_2(B)) \cap \text{FIRST}_2(d \cdot \text{FOLLOW}_2(B)) = \{aa, ac\} \cap \{da, d\$\} = \emptyset$

Gramatika je silná $LL(2)$.

Příklad

Následující gramatika generuje stejný jazyk jako ta, se kterou jsme pracovali v předchozím příkladu„ ale má jiná pravidla (vlastně vznikla z předchozí gramatiky ekvivalentními úpravami, postup si ukážeme později):

$G = (\{S, A, X, Y\}, \{a, b, c, d\}, P,\ S)$ s pravidly

$S \to aX \mid daY \mid \varepsilon$	$\text{FOLLOW}(S) = \{\$\}$
$A \to aA \mid \varepsilon$	$\text{FOLLOW}(A) = \{c, \$\}$
$X \to bAc \mid AcaY$	$\text{FOLLOW}(X) = \{\$\}$
$Y \to aAcA \mid dA$	$\text{FOLLOW}(Y) = \{\$\}$

Vytvoříme testovací množiny pro $k = 1$:

S:	$S \to aX$	$\text{FIRST}(aX \cdot \text{FOLLOW}(S)) = \{a\}$
	$S \to daY$	$\text{FIRST}(daY \cdot \text{FOLLOW}(S) = \{d\}$
	$S \to \varepsilon$	$\text{FIRST}(\varepsilon \cdot \text{FOLLOW}(S)) = \{\$\}$
A:	$A \to aA$	$\text{FIRST}(aA \cdot \text{FOLLOW}() = \{a\}$
	$A \to \varepsilon$	$\text{FIRST}(\varepsilon \cdot \text{FOLLOW}(A)) = \{c, \$\}$
X:	$X \to bAc$	$\text{FIRST}(bAc \cdot \text{FOLLOW}(X)) = \{b\}$
	$X \to AcaY$	$\text{FIRST}(AcaY \cdot \text{FOLLOW}(X)) = \{a, c\}$
Y:	$C \to aAcA$	$\text{FIRST}(aAcA \cdot \text{FOLLOW}(Y)) = \{a\}$
	$Y \to dA$	$\text{FIRST}(dA \cdot \text{FOLLOW}(Y)) = \{d\}$

Trochu praktičtější zápis, který můžeme použít v případě, že vzorec zvládáme „levou zadní“, je tento:

S: $\{a\},\ \{d\},\ \{\$\}$
A: $\{a\},\ \{c, \$\}$
X: $\{b\},\ \{a, c\}$
Y: $\{a\},\ \{d\}$

Pak platí, že množiny na tomtéž řádku (tedy příslušející pravidlům přepisujícím stejný neterminál) musí být vždy po dvou navzájem disjunktní. Podrobný zápis je následující:

$\text{FIRST}(aX \cdot \text{FOLLOW}(S)) \cap \text{FIRST}(daY \cdot \text{FOLLOW}(S)) = \{a\} \cap \{d\} = \emptyset$
$\text{FIRST}(aX \cdot \text{FOLLOW}(S)) \cap \text{FIRST}(\varepsilon \cdot \text{FOLLOW}(S)) = \{a\} \cap \{\$\} = \emptyset$
$\text{FIRST}(daY \cdot \text{FOLLOW}(S)) \cap \text{FIRST}(\varepsilon \cdot \text{FOLLOW}(S)) = \{d\} \cap \{\$\} = \emptyset$
$\text{FIRST}(aA \cdot \text{FOLLOW}(A)) \cap \text{FIRST}(\varepsilon \cdot \text{FOLLOW}(A)) = \{a\} \cap \{a, c\} = \emptyset$
$\text{FIRST}(bAc \cdot \text{FOLLOW}(X)) \cap \text{FIRST}(AcaY \cdot \text{FOLLOW}(X)) = \{b\} \cap \{a, c\} = \emptyset$
$\text{FIRST}(aAcA \cdot \text{FOLLOW}(Y)) \cap \text{FIRST}(dA \cdot \text{FOLLOW}(Y)) = \{a\} \cap \{d\} = \emptyset$

Gramatika je silná $LL(1)$.

Tentýž jazyk může být generován více různými gramatikami a každá z těchto gramatik může být silná LL(k) pro různá čísla k. Když určujeme typ jazyka a máme k dispozici více gramatik různého typu, vždy vybíráme ten „nejlepší“ případ, tedy gramatiku s nejmenším číslem k. Jazyk generovaný gramatikami z příkladů na stranách 55 a 56 je proto silný $LL(1)$, protože pro něj existuje LL gramatika pro $k = 1$.

Poznámka

Existují také $LL(0)$ gramatiky, ve kterých pro každý neterminál existuje právě jedno pravidlo (tj. nepotřebujeme pomoc ze vstupu při rozhodování mezi pravidly). Tyto gramatiky mohou generovat pouze jazyk s konečným počtem slov, protože zde není možná rekurze, a nejsou vhodné pro popis programovacích jazyků.

Úkol

U každé z následujících gramatik zjistěte, zda je silná $LL(1)$ nebo silná $LL(2)$. Vypisujte všechny množiny FOLLOW, FOLLOW_2 a testovací množiny pro jednotlivá pravidla.

$G_1 = (\{S, A, B\},\ \{a, b, c, d\},\ P,\ S)$
$S \to AbB \mid aBb$
$A \to abAA \mid c \mid \varepsilon$
$B \to daB \mid \varepsilon$

$G_2 = (\{S, A, B, C, D\},\ \{a, b, c, d, x, y, z\},\ P,\ S)$
$S \to aABbCD \mid \varepsilon$
$A \to ASd \mid \varepsilon$
$B \to SAc \mid xC \mid \varepsilon$
$C \to Sy \mid Cz \mid \varepsilon$
$D \to aBD \mid \varepsilon$

$G_3 = (\{S, A, B\},\ \{a, b, c, d\},\ P,\ S)$
$S \to acA \mid AaB$
$A \to \varepsilon \mid cAB$
$B \to bB \mid d$

3.6 $LL(1)$ překlady

3.6.1 $LL(1)$ gramatika

Jak bylo výše uvedeno, platí, že čím menší je číslo k, tím lépe. Už dříve jsme si ověřili, že sestrojení množin FIRST_k a FOLLOW_k pro větší čísla k je poněkud problematické, ale tím problémy nekončí – ono se to taky hůře programuje. Takže nejlepší je, pokud se nám podaří navrhnout silnou $LL(1)$ gramatiku. Jak je to vlastně se „silou“ $LL(1)$ gramatik?

Podle druhé definice $LL(k)$ gramatik v kapitole 3.5.1 na straně 54 by tedy $LL(1)$ gramatika měla být taková bezkontextová gramatika $G = (N, T, P, S)$, kde v případě existence dvou levých derivací
$S \Rightarrow^* wA\alpha \Rightarrow w\beta\alpha \Rightarrow^* wx$
$S \Rightarrow^* wA\alpha \Rightarrow w\gamma\alpha \Rightarrow^* wy\ ,\qquad$ kde $A \in N,\ w, x, y \in T^*,\ \alpha, \beta, \gamma \in (N \cup T)^*$
takových, že $\mathrm{FIRST}(x) = \mathrm{FIRST}(y)$, vždy platí $\beta = \gamma$. Bude lepší se zaměřit na ekvivalentní tvrzení, které najdeme v následné větě: pro každá dvě různá pravidla se stejnou levou stranou $A \to \beta \mid \gamma$ a každou levou derivaci $S \Rightarrow^* awA\alpha$ platí

$$\mathrm{FIRST}(\beta \cdot \alpha) \cap \mathrm{FIRST}(\gamma \cdot \alpha) = \emptyset \tag{3.19}$$

Tím se dostáváme do jedné ze dvou situací:

1. Pokud žádný z řetězců β a γ nelze přepsat pravidly gramatiky na ε, pak vzorec (3.19) můžeme zjednodušit – odebereme α:
$$\mathrm{FIRST}(\beta) \cup \mathrm{FIRST}(\gamma) = \emptyset \tag{3.20}$$
2. Pokud existuje možnost, že některý z řetězců β nebo γ lze přepsat na ε, pak musíme počítat s tím, že nám z větné formy „zmizí“, a tedy potřebujeme zjistit, co v dané větné formě bude následovat. Proto vzorec upravíme trochu jiným způsobem:
$$\mathrm{FIRST}(\beta \cdot \mathrm{FOLLOW}(A)) \cap \mathrm{FIRST}(\gamma \cdot \mathrm{FOLLOW}(A)) = \emptyset \tag{3.21}$$

Dá se říct, že první situace je speciálním případem té druhé, tedy nám stačí počítat s druhou situací. Co to jen připomíná? Ano, vzorec pro silnou $LL(k)$ gramatiku. K tomuto vzorci jsme dospěli z definice $LL(1)$ gramatiky, což znamená, že každá $LL(1)$ gramatika je silná, proto pro rozhodování mezi dvěma pravidly se stejnou levou stranou nám stačí jen jeden symbol ze vstupu a nemusíme se dívat hlouběji do zásobníku.

Věta

Každá $LL(1)$ gramatika je silná $LL(1)$ gramatika.

Důkaz: Důkaz této věty jsme si naznačili o něco výše. Vychází z toho, že všechny řetězce v množině FIRST jsou jednoznakové, což postupu rozhodování podle jednoho symbolu na vstupu dodává jednoznačnost. □

Připomeňme si, že pokud $LL(k)$ gramatika není silná (je slabá), pak nám při rozhodování mezi pravidly při konstrukci derivačního stromu nestačí pouze orientovat se podle zatím nezpracovaných symbolů na vstupu, ale rozhodujeme se i podle (zatím nezpracovaného) obsahu zásobníku, což při programování může znamenat komplikace.

Ovšem pozor, pro vyšší čísla k to neplatí, protože množiny FIRST_k mohou obsahovat i kratší řetězce než k.

Protože každá $LL(1)$ gramatika je silná, můžeme pro testování používat vzorec

$$\text{FIRST}(\alpha \cdot \text{FOLLOW}(A)) \ \cap \ \text{FIRST}(\beta \cdot \text{FOLLOW}(A)) \ = \ \emptyset \tag{3.22}$$

Příklad

Pro zpracování matematických výrazů se často používá gramatika, která má dvě dobré vlastnosti: je typu $LL(1)$ a navíc zachovává prioritu operátorů. Ta druhá vlastnost by se snad zdála u syntaktické analýzy zbytečná, má však svůj význam u sémantické analýzy, která je, jak zjistíme později, se syntaxí velmi těsně spojena.

$G = (\{S, A, B, C, D\},\ \{i, n, (,), +, -, *, /\},\ P,\ S)$ s pravidly

$S \to AB$	①
$B \to +AB \mid -AB \mid \varepsilon$	②,③,④
$A \to CD$	⑤
$D \to *CD \mid /CD \mid \varepsilon$	⑥,⑦,⑧
$C \to (S) \mid i \mid n$	⑨,⑩,⑪

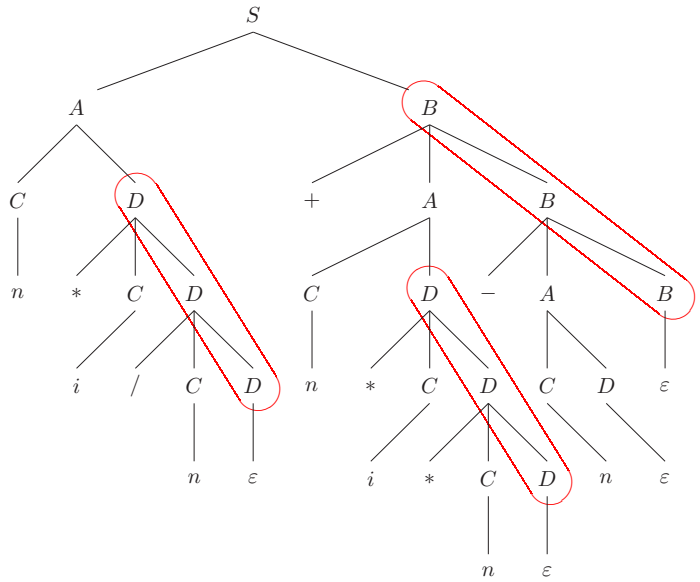
Derivace věty $n * i/n + n * i * n - n$ je následující:

$S \Rightarrow AB \Rightarrow CDB \Rightarrow nDB \Rightarrow n * CDB \Rightarrow n * iDB \Rightarrow n * i/CDB \Rightarrow n * i/nDB \Rightarrow n * i/nB \Rightarrow$
$\Rightarrow n * i/n + AB \Rightarrow n * i/n + CDB \Rightarrow n * i/n + nDB \Rightarrow n * i/n + n * CDB \Rightarrow n * i/n + n * iDB \Rightarrow$
$\Rightarrow n * i/n + n * i * CDB \Rightarrow n * i/n + n * i * nDB \Rightarrow n * i/n + n * i * nB \Rightarrow n * i/n + n * i * n - AB \Rightarrow$
$\Rightarrow n*i/n+n*i*n-CDB \Rightarrow n*i/n+n*i*n-nDB \Rightarrow n*i/n+n*i*n-nB \Rightarrow n*i/n+n*i*n-n$

Jedna z možných podob této věty s přidanou sémantikou by mohla být třeba $5 * x/2 + 78 * y * 21 - 45$.

Pokud vytvoříme derivační strom podle této derivace, zjistíme, že operátory s nižší prioritou $(+, -)$ tvoří hlavní větev shora dolů doprava, níže ve stejném směru jsou podvětve generující podvýrazy s operátory s vyšší prioritou $(*, /)$. Tento derivační strom můžeme vidět na obrázku 3.5 na straně 60 včetně vyznačení větví, ve kterých se generují skupiny podvýrazů s operátory se stejnou prioritou.

Vyhodnocování probíhá od listů (dole) ke kořeni, proto jsou nejdřív zpracovány operátory s vyšší prioritou a pak teprve operátory s nišší prioritou. Jedinou výjimku samozřejmě tvoří výrazy v závorkách (pravidlo ⑨), ty generují samostatné podstromy.



Obrázek 3.5: Derivační strom matematického výrazu v $LL(1)$ gramatice

3.6.2 Transformace na $LL(1)$ gramatiku

Pokud vytvoříme gramatiku, která není $LL(1)$, a přitom je tato vlastnost pro nás důležitá, potom se pokusíme ji transformovat na $LL(1)$ gramatiku. Možností transformace je mnoho, ale o žádné nemůžeme říci, že spolehlivě vede k cíli, tedy se jedná o nedeterministický postup.

Odstranění levé rekurze. Obecný tvar množiny pravidel s levou rekurzí je tento:
$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$

kde řetězce β_i nezačínají neterminálem A. Takovou množinu pravidel přepíšeme na dvě jiné množiny:
$A \to \beta_1 B \mid \beta_2 B \mid \ldots \mid \beta_m B$
$B \to \alpha_1 B \mid \alpha_2 B \mid \ldots \mid \alpha_n B \mid \varepsilon$

Jak původní pravidla, tak i ta nově vytvořená generují stejné výstupy – na začátku slova bude některý podřetězec β_i a následuje rekurzí vytvořená posloupnost podřetězců α_j. Levou rekurzi transformujeme na pravou, která nám při levé derivaci nevadí.

Příklad

Následující gramatiku s levou rekurzí transformujeme na $LL(1)$ gramatiku.
$G = (\{P, Q, R\},\ \{i, n, +, -, *, /, (,)\},\ P_G,\ P)$ s pravidly v množině P_G:
$P \to P + Q \mid P - Q \mid Q$
$Q \to Q * R \mid Q/R \mid R$
$R \to (P) \mid i \mid n$

Odstraníme levou rekurzi v pravidlech $P \to P + Q \mid P - Q \mid Q$:
$P \to QU$
$U \to +QU \mid -QU \mid \varepsilon$

Podobně zpracujeme také pravidla pro symbol Q:
$Q \to RV$
$V \to *RV \mid /RV \mid \varepsilon$

Získali jsme tuto gramatiku:
$G' = (\{P, U, Q, V, R\},\ \{i, n, +, -, *, /, (,)\},\ P'_G,\ P)$ s pravidly v množině P'_G:
$P \to QU$
$U \to +QU \mid -QU \mid \varepsilon$
$Q \to RV$
$V \to *RV \mid /RV \mid \varepsilon$
$R \to (P) \mid i \mid n$

Jak vidíme, vytvořená gramatika je ekvivalentní (až na označení neterminálů) s $LL(1)$ gramatikou z příkladu na straně 59, tedy ani nemusíme testovat.

Faktorizace pravidel. Tuto metodu (*levá faktorizace*) použijeme, jestliže několik pravidel pro přepis téhož neterminálu začíná stejným řetězcem terminálů, tedy pravidlo je
$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$
V takovém případě „vytkneme“ shodné části pravidel (zde α) a zavedeme nový neterminál (zde B):
$A \to \alpha B$
$B \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Je to podobné, jako když v aritmetice zpracováváme matematické výrazy se závorkami.

Příklad

Následující gramatiku transformujeme na $LL(1)$.

$G = (\{S, A, B\},\ \{a, b, c\},\ P,\ S)$ s pravidly

$S \to AaBB \mid AaS$	$\text{FOLLOW}(S) = \{\$\}$
$A \to bA \mid \varepsilon$	$\text{FOLLOW}(A) = \{a\}$
$B \to cdB \mid c$	$\text{FOLLOW}(B) = \{c, \$\}$

Je třeba provést faktorizaci pravidel pro neterminály S a B. Pravidla přepisující neterminál S začínají podřetězcem Aa, proto je upravíme takto:
$S \to AaC$
$C \to BB \mid S$

Úprava pravidel přepisujících B je podobná, začínají podřetězcem c, který „vytkneme“. Získali jsme tuto gramatiku:

$G' = (\{S, A, B, C, D\},\ \{a, b, c\},\ P',\ S)$ s pravidly

$S \to AaC$	$\text{FOLLOW}(S) = \{\$\}$	$\text{FIRST}(BB) = \{c\}$
$C \to BB \mid S$	$\text{FOLLOW}(C) = \{\$\}$	$\text{FIRST}(S) = \{b, a\}$
$A \to bA \mid \varepsilon$	$\text{FOLLOW}(A) = \{a\}$	
$B \to cD$	$\text{FOLLOW}(B) = \{c, \$\}$	
$D \to dB \mid \varepsilon$	$\text{FOLLOW}(D) = \{c, \$\}$	

Tato gramatika je již $LL(1)$. Je zajímavé, že transformace neměla prakticky žádný vliv na množiny FOLLOW, dokonce množiny FOLLOW přidaných neterminálů kopírují množiny těch neterminálů, pro které byly vytvořeny.

Eliminace pravidel. Někdy pomůže dosadit do pravidel za některé neterminály jejich pravé strany (což může příznivě ovlivnit množiny FOLLOW).

Tento postup bohužel moc často k cíli nevede, protože většinou je nutné následně použít faktorizaci, která nás ovšem dostane do původního stavu, kdy gramatika nebyla $LL(1)$.

Postup eliminace pravidel si ukážeme v souhrnném příkladu s demonstrací několika různých typů úprav.

Redukce množin FOLLOW. Je-li pro některý neterminál příčina kolize v množině FOLLOW, může pomoci přidání nového neterminálu, jehož množina FOLLOW převezme část prvků množiny FOLLOW konfliktního neterminálu.

Příklad

Následující gramatiku transformujeme na $LL(1)$.

$G = (\{S, A, B\},\ \{a, b, c, d\},\ P,\ S)$ s pravidly

$S \to aA \mid BA$	$\text{FOLLOW}(S) = \{\$\}$
$A \to bcd$	$\text{FOLLOW}(A) = \{b, \$, c\}$
$B \to baBcAA \mid \varepsilon$	$\text{FOLLOW}(B) = \{b, c\}$

Tato gramatika není $LL(1)$, ke konfliktu dochází u neterminálu B:
$\text{FIRST}(baBcAA \cdot \text{FOLLOW}(B)) \cap \text{FIRST}(\varepsilon \cdot \text{FOLLOW}(B)) = \{b\}$

Při konfliktu souvisejícím s množinami FOLLOW je třeba zjistit, jak se konfliktní terminál do příslušné množiny dostal. V našem případě byl symbol b do $\text{FOLLOW}(B)$ zařazen při vyhodnocení pravidla $S \to BA$, proto budeme transformovat právě toto pravidlo.

Použijeme metodu redukce množiny FOLLOW, která se provádí obvykle „pohlcením“ konfliktního terminálu jemu předcházejícím neterminálem. Zde však za neterminálem B nemáme přímo b, proto předem provedeme jinou úpravu – dosazení pravé strany pravidla za následný symbol A. Pravidlo bude vypadat takto:
$S \to Bbcd$

Teď máme přímo za B terminál b, který má být „pohlcen“. V nahrazovaném pravidle vytvoříme nový neterminál P, který bude představovat předchozí řetězec Bb, a v novém pravidle ho přepíšeme ne přímo na tento řetězec Bb, ale symbol B rozvineme:
$S \to aA \mid Pcd$
$P \to baBcAAb \mid b$

Při rozvinutí symbolu B podle pravidel tento symbol přepisujících nesmíme zapomenout na konec přidat také „pohlcený“ terminál b.

Po úpravách je třeba ještě použít faktorizaci v pravidlech pro neterminál P, výsledná gramatika tedy bude následující:

$G' = (\{S, A, B, C, P\},\ \{a, b, c, d\},\ P',\ S)$ s pravidly

$S \to aA \mid Pcd$	$\text{FOLLOW}(S) = \{\$\}$
$P \to bC$	$\text{FOLLOW}(P) = \{c\}$
$C \to aBcAAb \mid \varepsilon$	$\text{FOLLOW}(C) = \{c\}$
$A \to bcd$	$\text{FOLLOW}(A) = \{b, \$, c\}$
$B \to baBcAA \mid \varepsilon$	$\text{FOLLOW}(B) = \{c\}$

Tato gramatika je $LL(1)$, také díky odstranění symbolu b z množiny FOLLOW(B).

Všechny zde uvedené transformace jsou *ekvivalentními úpravami gramatik*, to znamená, že původní a upravená gramatika generují tentýž jazyk. Ekvivalence platí na úrovni syntaxe, v případě sémantiky je vhodné gramatiku prověřit a znovu určit, které sémantické operace se kdy mají provést. K sémantice se ovšem dostaneme až později.

Příklad

V jednom z příkladů jsme narazili na gramatiku, která není $LL(1)$ (str. 55), a hned v následujícím příkladu (str. 56) byla gramatika k ní ekvivalentní (tj. generující stejný jazyk), ale splňující vlastnosti $LL(1)$ gramatiky. Nyní si ukážeme, jak tu původní transformovat na druhou uvedenou gramatiku.

Následující gramatika tedy není $LL(1)$:

$G_1 = (\{S, A, B\},\ \{a, b, c, d\},\ P_1,\ S)$

$S \to abAc \mid BaBA \mid \varepsilon$	FOLLOW(S) = $\{\$\}$	S: $\{a\}$, $\{a, d\}$, $\{\$\}$
$A \to aA \mid \varepsilon$	FOLLOW(A) = $\{c, \$\}$	A: $\{a\}$, $\{c, \$\}$
$B \to aAc \mid d$	FOLLOW(B) = $\{a, \$\}$	B: $\{a\}$, $\{d\}$

Nejdřív použijeme eliminaci pravidel pro neterminál B (vlastně tím eliminujeme celý neterminál):

$G_2 = (\{S, A\},\ \{a, b, c, d\},\ P_2,\ S)$

$S \to abAc \mid aAcaaAcA \mid aAcadA \mid daaAcA \mid dadA \mid \varepsilon$

$A \to aA \mid \varepsilon$

Eliminaci nemusíme provádět až do důsledků. Pokud však nahradíme jen některé výskyty daného neterminálu pravými stranami (postupně všech) jeho pravidel, pak z gramatiky nesmíme dosazovaná pravidla odstranit.

původní:	B	a	B	A
varianta 1:	aAc	a	aAc	A
varianta 2:	aAc	a	d	A
varianta 3:	d	a	aAc	A
varianta 4:	d	a	d	A

Tabulka 3.2: Ukázka použití eliminace pravidel

Zbavili jsme se dvou pravidel pro neterminál B, ale zato jedno z pravidel pro neterminál S jsme nahradili čtyřmi jinými. Je to proto, že v původním pravidle $S \to BaBA$ se symbol B vyskytoval dvakrát, zároveň pro symbol B jsou dvě pravidla, a tedy je třeba vytvořit všechny možné varianty (vlastně nasimulovat zpracování obou výskytů symbolu B různými pravidly). Vytvoření různých variant původního pravidla vidíme v tabulce 3.2.

Ovšem tato gramatika ještě není $LL(1)$. Tři pravidla přepisující symbol S začínají stejným podřetězcem a, další dvě pravidla začínají podřetězcem da. V těchto případech by se testovací množiny dostaly do kolize. Tedy provedeme faktorizaci, přičemž nás zajímají tato pravidla:

$S \to abAc \mid aAcaaAcA \mid aAcadA \mid daaAcA \mid dadA \mid \varepsilon$

Vytvoříme tedy dva nové neterminály:

$G_3 = (\{S, A, X, Y\},\ \{a, b, c, d\},\ P_3,\ S)$

$S \to aX \mid daY \mid \varepsilon$

$A \to aA \mid \varepsilon$

$X \to bAc \mid AcaaAcA \mid AcadA$

$Y \to aAcA \mid dA$

Tím úpravy nekončí. Všimněte si, že v pravidlech nového neterminálu X máme taky kolizi (v druhém a třetím pravidle), tedy opět faktorizujeme.

$G_4 = (\{S, A, X, Y, Z\},\ \{a, b, c, d\},\ P_4,\ S)$
$S \to aX \mid daY \mid \varepsilon$
$A \to aA \mid \varepsilon$
$X \to bAc \mid AcaZ$
$Y \to aAcA \mid dA$
$Z \to aAcA \mid dA$

Zbývá poslední úprava – podívejte se na poslední dva řádky (pravidla pro neterminály Y a Z). Jsou naprosto stejná. Mohli bychom provést kompletní redukci gramatiky, ale postačí nám nahrazení neterminálu Z ekvivalentním neterminálem Y. Odstraníme pravidla pro neterminál Z a všechny výskyty symbolu Z na pravých stranách pravidel nahradíme neterminálem Y.

$G_5 = (\{S, A, X, Y\}, \{a, b, c, d\}, P,\ S)$

$S \to aX \mid daY \mid \varepsilon$	$\text{FOLLOW}(S) = \{\$\}$	S: $\{a\},\ \{d\},\ \{\$\}$
$A \to aA \mid \varepsilon$	$\text{FOLLOW}(A) = \{c, \$\}$	A: $\{a\},\ \{c, \$\}$
$X \to bAc \mid AcaY$	$\text{FOLLOW}(X) = \{\$\}$	X: $\{b\},\ \{a, c\}$
$Y \to aAcA \mid dA$	$\text{FOLLOW}(Y) = \{\$\}$	Y: $\{a\},\ \{d\}$

To je přesně ta gramatika, kterou jsme testovali na straně 56 a zjistili jsme, že je $LL(1)$.

Úkoly

1. S touto gramatikou jste se už setkali v úkolech na straně 57, kde jste zřejmě zjistili, že není $LL(1)$:

   $G_3 = (\{S, A, B\},\ \{a, b, c, d\},\ P,\ S)$
   $S \to acA \mid AaB$
   $A \to \varepsilon \mid cAB$
   $B \to bB \mid d$

   Tuto gramatiku však lze transformovat na $LL(1)$. Proveďte tuto transformaci.

   Nápověda: v druhém pravidle pro neterminál S nahraďte symbol A pravými stranami pravidel pro tento neterminál, čímž z jednoho pravidla uděláte dvě, ale pravidla pro A neodstraňujte, protože tento symbol se vyskytuje i jinde (tedy proveďte částečnou eliminaci). Pak faktorizujte.

2. S následující gramatikou jsme se setkali v úkolu na straně 53, kde jsme měli vypočíst pomocné množiny pro $k = 1$ a $k = 2$.

   $G = (\{S, A, B, C\},\ \{a, b, c, d\},\ P,\ S)$ s pravidly
   $S \to abA \mid cbBC$
   $A \to cA \mid dB \mid a$
   $B \to bBS \mid a$
   $C \to cC \mid cBd \mid \varepsilon$

   Ověřte, že tato gramatika není LL(1), a zvažte, jestli by byla možná transformace.

3.6.3 Překladový automat

Účelem překladu ovšem není jen popis syntaktické struktury, tedy gramatika. Potřebujeme postup, jakým z již existujícího vstupu dostaneme příslušný výstup (v našem případě zatím posloupnost čísel použitých pravidel). Zatímco gramatika je generativní struktura, automat slouží k rozpoznávání vstupu.

Nejdřív na příkladu připomeneme postup vytvoření zásobníkového automatu podle gramatiky, který známe z předmětu *Teorie jazyků a automatů*, tedy vytvoření „obyčejného" zásobníkového automatu, který pouze rozpoznává vstup a nic nepřekládá. Zatím to tedy není překladový automat, ale běžný zásobníkový, vytvořený podle gramatiky z příkladu na straně 59.

Příklad

Vytvoříme zásobníkový automat rozpoznávající slova jazyka generovaného gramatikou
$G = (\{S, A, B, C, D\},\ \{i, n, (,), +, -, *, /\},\ P,\ S)$ s pravidly

$S \to AB$	①
$B \to +AB \mid -AB \mid \varepsilon$	②,③,④
$A \to CD$	⑤
$D \to *CD \mid /CD \mid \varepsilon$	⑥,⑦,⑧
$C \to (S) \mid i \mid n$	⑨,⑩,⑪

Vytváříme tedy automat $\mathcal{A} = (\{q\},\ T,\ T \cup N,\ \delta,\ q,\ S,\ \emptyset)$, funkce δ je definovaná následovně:

1. Pro všechna pravidla ve tvaru $A \to \alpha$ bude $\delta(q, \varepsilon, A) \ni (q, \alpha)$:
   $\delta(q, \varepsilon, S) = \{(q, AB)\}$
   $\delta(q, \varepsilon, B) = \{(q, +AB), (q, -AB), (q, \varepsilon)\}$
   $\delta(q, \varepsilon, A) = \{(q, CD)\}$
   $\delta(q, \varepsilon, D) = \{(q, *CD), (q, /CD), (q, \varepsilon)\}$
   $\delta(q, \varepsilon, C) = \{(q, (S)), (q, i), (q, n)\}$
2. Pro všechny symboly $a \in T$ přidáme $\delta(q, a, a) = \{(q, \varepsilon)\}$:

$\delta(q, +, +) = \{(q, \varepsilon)\}$	$\delta(q, (, () = \{(q, \varepsilon)\}$
$\delta(q, -, -) = \{(q, \varepsilon)\}$	$\delta(q,),)) = \{(q, \varepsilon)\}$
$\delta(q, *, *) = \{(q, \varepsilon)\}$	$\delta(q, n, n) = \{(q, \varepsilon)\}$
$\delta(q, /, /) = \{(q, \varepsilon)\}$	$\delta(q, i, i) = \{(q, \varepsilon)\}$

První bod postupu říká, že pokud je na vrcholu zásobníku neterminál, vyjmeme ho a nahradíme pravou stranou některého pravidla pro tento symbol. Například pro pravidlo $A \to \alpha$ – když ze zásobníku vyjmeme A, do zásobníku naskládáme řetězec α.

Pokud pro neterminál existuje více pravidel (B, C, D), je třeba se mezi nimi rozhodnout. Máme k dispozici množiny FIRST pravé strany pravidel – podíváme se na první symbol nepřečtené části vstupu (stačí nám pouze jeden, je to $LL(1)$ gramatika). Protože pravidla mají navzájem disjunktní množiny FIRST (to plyne z definice $LL(1)$ gramatiky), je rozhodování jednoznačné.

Druhý bod určuje, jaká je reakce automatu v konfiguraci, kdy je na vrcholu zásobníku terminální symbol. Tento symbol vyjme, srovná se vstupem, a pokud jsou oba symboly stejné, posune se na vstupní pásce na další symbol a pokračuje ve výpočtu.

Počáteční konfigurace je (q, w, S), kde $w \in T^*$ je některé vstupní slovo.

Automat vytvořený podle postupu z předchozího příkladu rozpoznává věty jazyka, ale

- je nedeterministický, rozhodování pomocí množin FIRST není zahrnuto ve funkci δ, třebaže naprogramovat se nějak musí,
- nemáme kam zapsat výstup – v našem případě zatím levý rozklad,
- chybí možnost okamžitě zjistit chybu, podle funkce δ to může být se zpožděním.

Proto zásobníkový automat trochu upravíme a dovybavíme, vytvoříme překladový automat. Bude pracovat na podobném principu jako klasický zásobníkový automat (použijeme zásobník, v něm budeme vždy levou stranu pravidla nahrazovat pravou stranou), navíc přidáme možnost deterministicky se rozhodovat podle vstupu v případě, že pro tentýž symbol na vrcholu zásobníku existuje více možných akcí, a samozřejmě přidáme výstupní pásku.

Definice (Překladový automat pro $LL(1)$ překlad)

Překladový automat $LL(1)$ gramatiky $G = (N, T, P, S)$ je zásobníkový automat s jediným stavem rozšířený o výstupní pásku a definovaný rozkladovou tabulkou.

Konfigurace překladového automatu má tvar (α, β, γ), kde $\alpha \in T^* \cdot \$$ je nepřečtená část vstupní pásky, $\beta \in (N \cup T)^* \cdot \#$ je obsah zásobníku a γ obsah výstupní pásky. *Počáteční konfigurace* je $(w\$, S\#, \varepsilon)$, kde w je vstupní řetězec, S startovací symbol gramatiky G a $\#$ symbol konce zásobníku.

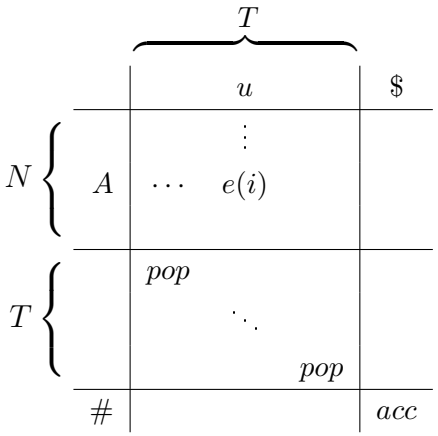
Rozkladová tabulka automatu pro LL(1) gramatiku je zobrazení
$M : (T \cup N \cup \{\#\}) \times (T \cup \{\$\}) \mapsto \{expand(1), \ldots, expand(n), pop, accept, error\}$,
kde jednotlivé funkční hodnoty mají tento význam:

- $expand(i)$, $1 \leq i \leq n$: Je-li $A \to \alpha$ i-té pravidlo gramatiky, na vrcholu zásobníku je neterminál A, na vstupu symbol x, v tabulce je $M[A, x] = expand(i)$, provede automat změnu konfigurace $(x\sigma, A\phi, \gamma) \vdash (x\sigma, \alpha\phi, \gamma i)$, tedy v zásobníku nahradí symbol A pravou stranu pravidla $A \to \alpha$, vstupní pásky si nevšímá a na výstupní pásku připíše číslo i (obdoba prvního kroku v předchozím postupu).
- *pop*: Je-li na vrcholu zásobníku a na vstupní pásce tentýž terminální symbol x, provede automat změnu konfigurace $(x\sigma, x\phi, \gamma) \vdash (\sigma, \phi, \gamma)$, tedy odstraní symbol x z vrcholu zásobníku a na vstupní pásce se posune na další znak (obdoba druhého kroku předchozího postupu).
- *accept*: K přijetí vstupu dojde, pokud je zásobník prázdný a vstup celý přečtený. Na výstupní pásce je levý rozklad vstupní věty.
- *error*: Tato hodnota znamená chybu ve výpočtu, zde odpovídá syntaktické chybě.

Rozkladovou tabulku tvoříme takto:

1. Řádky ohodnotíme všemi symboly, které mohou být v zásobníku, tedy neterminály, terminály a symbolem konce zásobníku. Sloupce ohodnotíme všemi symboly, které mohou být na vstupu, tedy terminály a symbolem konce vstupu \$ (odpovídá konci souboru nebo poslednímu ukazateli dynamického seznamu).
2. Postupně probíráme pravidla gramatiky. Pro každé pravidlo $A \to \alpha$ (i-té pravidlo gramatiky) postupujeme takto:
   - vypočteme množinu $U = \text{FIRST}(\alpha \cdot \text{FOLLOW}(A))$,
   - v tabulce v řádku označeném A ve všech sloupcích označených symboly z množiny U doplníme funkci $expand(i)$.

3. V řádcích označených terminálními symboly napíšeme po diagonále funkci *pop*, a to tak, že platí $M[x, x] = pop$, kde x je terminální symbol (v zásobníku je tentýž symbol jako na vstupu).
4. V buňce tabulky $M[\#, \$]$ bude *accept* (konec zásobníku, konec vstupu).
5. Ostatní, dosud nevyplněné buňky obsahují hodnotu *error*.



$A \to \alpha$ je i-té pravidlo gramatiky
$u \in \text{FIRST}(\alpha \cdot \text{FOLLOW}(A))$

Zásobník	Vstup
	Akce: • *expect* • *pop* • *accept* • *error*

Tabulka 3.3: Schémata rozkladové tabulky pro $LL(1)$ gramatiku

V tabulce 3.3 na straně 67 je postup zobrazen na schématech. Názvy funkcí *expand* a *accept* jsou zkráceny na *e* a *acc*, v prázdných buňkách tabulky je funkce *error*. Podle schématu vidíme, že

- horní část tabulky ošetřuje případy, kdy na vrcholu zásobníku najdeme neterminál (pak provedeme expanzi podle určitého pravidla přepisujícího tento neterminál, konkrétní pravidlo je předepsáno argumentem funkce pro expanzi),
- kdežto spodní část tabulky ošetřuje případy, kdy na vrcholu zásobníku je terminál (na což reagujeme kontrolou, zda je na vstupu tentýž terminál, a posunem na vstupu o jeden symbol dál) nebo symbol konce zásobníku (v případě, že je celý vstup přečtený, akceptujeme, jinak hlásíme chybu).

Vytvoření rozkladové tabulky a práci s ní ukážeme na gramatice z předchozích příkladů, o které již víme, že je typu $LL(1)$. Na výstupní pásce bude posloupnost čísel. Aby nedošlo ke „slévání“ číslic různých čísel v konfiguraci, budeme čísla oddělovat čárkou (s menšími mezerami za čárkami), třebaže toto značení není korektní vzhledem k definici.

Příklad

K dané gramatice vytvoříme rozkladovou tabulku.

$S \to aB \mid daC \mid \varepsilon$	①,②,③	$\text{FOLLOW}(S) = \{\$\}$
$A \to aA \mid \varepsilon$	④,⑤	$\text{FOLLOW}(A) = \{c, \$\}$
$B \to bAc \mid AcaC$	⑥,⑦	$\text{FOLLOW}(B) = \{\$\}$
$C \to aAcA \mid dA$	⑧,⑨	$\text{FOLLOW}(C) = \{\$\}$

Vypočteme množiny $\text{FIRST}(\alpha \cdot \text{FOLLOW}(A))$ pro všechna pravidla $A \to \alpha$. Tyto množiny využijeme jak při testování, zda je gramatika typu $LL(1)$, tak i při konstrukci rozkladové tabulky.

$\text{FIRST}(aB \cdot \text{FOLLOW}(S)) = \{a\}$
$\text{FIRST}(daC \cdot \text{FOLLOW}(S)) = \{d\}$
$\text{FIRST}(\varepsilon \cdot \text{FOLLOW}(S)) = \{\$\}$
$\text{FIRST}(aA \cdot \text{FOLLOW}(A)) = \{a\}$
$\text{FIRST}(\varepsilon \cdot \text{FOLLOW}(A)) = \{c, \$\}$

$\text{FIRST}(bAc \cdot \text{FOLLOW}(B)) = \{b\}$ $\quad$ $\text{FIRST}(aAcA \cdot \text{FOLLOW}(C)) = \{a\}$
$\text{FIRST}(AcaC \cdot \text{FOLLOW}(B)) = \{a, c\}$ $\quad$ $\text{FIRST}(dA \cdot \text{FOLLOW}(C)) = \{d\}$

Otestujeme, zda je gramatika $LL(1)$:
$\text{FIRST}(aB \cdot \text{FOLLOW}(S)) \cap \text{FIRST}(daC \cdot \text{FOLLOW}(S)) = \emptyset$
$\text{FIRST}(aB \cdot \text{FOLLOW}(S)) \cap \text{FIRST}(\varepsilon \cdot \text{FOLLOW}(S)) = \emptyset$
$\text{FIRST}(daC \cdot \text{FOLLOW}(S)) \cap \text{FIRST}(\varepsilon \cdot \text{FOLLOW}(S)) = \emptyset$
$\text{FIRST}(aA \cdot \text{FOLLOW}(A)) \cap \text{FIRST}(\varepsilon \cdot \text{FOLLOW}(A)) = \emptyset$
$\text{FIRST}(bAc \cdot \text{FOLLOW}(B)) \cap \text{FIRST}(AcaC \cdot \text{FOLLOW}(B)) = \emptyset$
$\text{FIRST}(aAcA \cdot \text{FOLLOW}(C)) \cap \text{FIRST}(dA \cdot \text{FOLLOW}(C)) = \emptyset$

	a	b	c	d	\$
S	e1			e2	e3
A	e4		e5		e5
B	e7	e6	e7		
C	e8			e9	
a	*pop*				
b		*pop*			
c			*pop*		
d				*pop*	
#					*acc*

Všechny množiny, které jsme vytvořili z pravidel gramatiky, použijeme pro označení sloupců rozkladové tabulky.

Výraz *expand* zkracujeme na *e*, výraz *accept* zkracujeme na *acc*.

Podle rozkladové tabulky zpracujeme větu *acad*, počáteční konfigurace pro toto slovo je $(acad\$, S\#, \varepsilon)$:
$(acad\$,\ S\#,\ \varepsilon) \vdash (acad\$,\ aB\#,\ 1) \vdash$
$\vdash (cad\$,\ B\#,\ 1) \vdash (cad\$,\ AcaC\#,\ 1,7) \vdash$
$\vdash (cad\$,\ caC\#,\ 1,7,5) \vdash (ad\$,\ aC\#,\ 1,7,5) \vdash$
$\vdash (d\$\ ,C\#,\ 1,7,5) \vdash (d\$,\ dA\#,\ 1,7,5,9) \vdash$
$\vdash (\$,\ A\#,\ 1,7,5,9) \vdash (\$,\ \#,\ 1,7,5,9,5)$

Na konci výpočtu je celý vstup přečtený (zbývá tam jen symbol konce vstupu \$) a v zásobníku je pouze symbol konce zásobníku (#), proto věta *acad* patří do jazyka rozpoznávaného automatem a na výstupní pásce je levý rozklad pro tuto větu, posloupnost ①, ⑦, ⑤, ⑨, ⑤.

Protože spodní část tabulky pro $LL(1)$ gramatiku je vždy stejná, můžeme tabulku zkrátit a uvést pouze řádky označené neterminály.

Příklad

Sestrojíme rozkladovou tabulku ke gramatice generující matematické výrazy z příkladu na straně 59.

$S \to AB$	①	$\text{FOLLOW}(S) = \{\$,)\}$
$A \to CD$	②	$\text{FOLLOW}(A) = \{+, -, \$,)\}$
$B \to +AB \mid -AB \mid \varepsilon$	③, ④, ⑤	$\text{FOLLOW}(B) = \{\$,)\}$
$C \to (S) \mid i \mid n$	⑥, ⑦, ⑧	$\text{FOLLOW}(C) = \{*, /, +, -, \$,)\}$
$D \to *CD \mid /CD \mid \varepsilon$	⑨, ⑩, ⑪	$\text{FOLLOW}(D) = \{+, -, \$,)\}$

	i	n	$+$	$-$	$*$	$/$	(	)	\$
S	e1	e1					e1		
A	e2	e2					e2		
B			e3	e4				e5	e5
C	e7	e8					e6		
D			e11	e11	e9	e10		e11	e11

Podle tabulky zpracujeme větu $n + i * n$:

$(n + i * n\$,\ S\#,\ \varepsilon) \vdash (n + i * n\$,\ AB\#,\ 1) \vdash (n + i * n\$,\ CDB\#,\ 1,2) \vdash$
$\vdash (n + i * n\$,\ nDB\#,\ 1,2,8) \vdash (+i * n\$,\ DB\#,\ 1,2,8) \vdash (+i * n\$,\ B\#,\ 1,2,8,11) \vdash$
$\vdash (+i * n\$,\ +AB\#,\ 1,2,8,11,3) \vdash (i * n\$,\ AB\#,\ 1,2,8,11,3) \vdash$
$\vdash (i * n\$,\ CDB\#,\ 1,2,8,11,3,2) \vdash (i * n\$,\ iDB\#,\ 1,2,8,11,3,2,7) \vdash$
$\vdash (*n\$,\ DB\#,\ 1,2,8,11,3,2,7) \vdash (*n\$,\ *CDB\#,\ 1,2,8,11,3,2,7,9) \vdash$
$\vdash (n\$,\ CDB\#,\ 1,2,8,11,3,2,7,9) \vdash (n\$,\ nDB\#,\ 1,2,8,11,3,2,7,9,8) \vdash$
$\vdash (\$,\ DB\#,\ 1,2,8,11,3,2,7,9,8) \vdash (\$,\ B\#,\ 1,2,8,11,3,2,7,9,8,11) \vdash$
$\vdash (\$,\ \#,\ 1,2,8,11,3,2,7,9,8,11,5)$

Levý rozklad pro větu $n + i * n$ je ①, ②, ⑧, ⑪, ③, ②, ⑦, ⑨, ⑧, ⑪, ⑤.

Úkoly

1. Zjistěte, zda je následující gramatika $LL(1)$, a pokud ano, sestrojte rozkladovou tabulku. Podle gramatiky vygenerujte jakékoliv slovo délky alespoň 4 znaky, toto slovo pak zpracujte podle rozkladové tabulky.

   $G = (\{S, A, B\},\ \{a, b, c, d, m\},\ P,\ S)$ s pravidly
   $S \to aAB \mid bBS \mid \varepsilon$
   $A \to cBS \mid \varepsilon$
   $B \to dB \mid m$

2. V úkolech na straně 43 je následující gramatika:

   $G = (\{S, A, B, D, M, P, V\},\ \{b, e, ., ;, c, p, <, >, i, t, w, r, =, +, -, *\},\ R,\ S)$
   $S \to DbAe.$
   $A \to P\,;A \mid \varepsilon$
   $B \to c \mid p$
   $D \to p\,;D \mid \varepsilon$
   $M \to B < B \mid B = B$
   $P \to iMtP \mid wV \mid rp \mid p = V$
   $V \to B + B \mid B - B \mid B * B$

   Původně bylo naším úkolem sestrojení množin FIRST a FOLLOW. Ověřte, zda tato gramatika je $LL(1)$. Pokud ne, transformujte ji na $LL(1)$.

   Podle upravené gramatiky vytvořte derivaci jakéhokoliv slova w o délce alespoň 4. Následně sestrojte rozkladovou tabulku a podle této tabulky zpracujte slovo, které jste dříve podle gramatiky vygenerovali. Porovnejte derivaci podle gramatiky a odvození podle tabulky, všimněte si především obsahu zásobníku v konfiguracích.

3.6.4 Implementace metodou přepisu rozkladové tabulky

Implementaci – naprogramování – $LL(1)$ překladu provádíme na základě $LL(1)$ gramatiky, vypočtených množin FIRST a FOLLOW a případně také rozkladové tabulky. Může se to zdát zbytečně komplikované, ale výsledný program pracuje s poměrně dobrou časovou složitostí.

Při implementaci metodou přepisu rozkladové tabulky potřebujeme zásobník, do kterého na začátku programu vložíme symbol konce zásobníku # a startovací symbol gramatiky.

Budeme programovat překladač, ve kterém jsou všechny tři analýzy (lexikální, syntaktická a sémantická) v jednom průchodu, řídící fází bude syntaktická analýza. Předpokládejme, že máme funkci `Lex()` plnící roli lexikálního analyzátoru, která při každém zavolání vrátí jeden symbol.

Inicializační část postupu zahrnuje:

- vytvoření zásobníku a jeho prvotní naplnění symboly # a S, což známe z počáteční konfigurace,
- přednačtení jednoho symbolu ze vstupu (před každým krokem syntaktické analýzy chceme přednačtený symbol, který budeme porovnávat s již známými množinami $\mathrm{FIRST}(\alpha \cdot \mathrm{FOLLOW}(A))$), tedy jednou zavoláme funkci `Lex()`.

Funkce `Lex()` při každém zavolání uloží následující symbol ze vstupu do proměnné `symbol`, jak jsme si zavedli v kapitole o lexikální analýze. Buď jde o globální proměnnou, nebo je tato proměnná jiným způsobem zpřístupňována různým fázím překladu, případně předávána v parametrech funkce či vlastnostech objektu.

Účelem činnosti překladového automatu je nasimulovat si v zásobníku derivaci řetězce, který má na vstupu, čímž si konstruuje derivační strom tohoto řetězce. Derivační strom (ve formě levého rozkladu, tedy posloupnosti čísel použitých pravidel) bude na výstupu automatu.

Protože používáme levou derivaci, stačí během simulace v zásobníku udržovat jen tu část větné formy, která ještě není zpracovaná, kdežto terminální část v levé části větné formy (tj. už zpracovanou) už potřebovat nebudeme.

Pokud v derivaci nelze dále pokračovat tak, aby byl vygenerován přesně takový řetězec, jaký je na vstupu (tj. v rozkladové tabulce se dostaneme do prázdné buňky = `error`), znamená to, že ve vstupním řetězci je syntaktická chyba, a to na místě, kde se výpočet zastavil. Podle pravidla, které je právě vyhodnocováno, nahlásíme chybu (například může jít o pravidlo reprezentující část matematického výrazu nebo konkrétní příkaz programovacího jazyka).

 Při programování syntaktické analýzy metodou přepisu rozkladové tabulky budeme postupovat stejně, jak postupuje člověk při používání „papírové“ verze rozkladové tabulky. Potřebujeme tyto funkce odpovídající příslušným akcím z rozkladové tabulky:

- `expand(číslo_pravidla)` provede expanzi pravidla s daným číslem v zásobníku (místo neterminálu z levé strany pravidla, který byl už před voláním této funkce odstraněn, uloží do zásobníku pravou stranu pravidla) a na výstup přidá číslo pravidla,
- `pop()` ověří shodnost symbolu na vstupu se symbolem vyjmutým ze zásobníku a načte další symbol ze vstupu (tj. zavolá funkci `Lex()`),
- `accept()` při konci vstupu a konci zásobníku ukončí výpočet programu, zde jen jednoduše nastaví proměnnou `Konec` na `true`,
- `error(...)` ošetří chybu, která se vyskytla při překladu; konkretizaci chyby můžeme provést v parametru této funkce.
- `Akce()` provádí právě to, co děláme my osobně, když pracujeme s „papírovou“ rozkladovou tabulkou, tedy
  - vyjme ze zásobníku jeden symbol, tím určí řádek tabulky a podle symbolu na vstupu určí sloupec tabulky,
  - podle obsahu buňky na daném řádku a sloupci zavolá funkci `expand()`, `pop()`, `accept()` nebo `error()` (tu pro prázdnou buňku),

  - je volána v cyklu tak dlouho, dokud není konec zpracovávaného programu (tj. dokud proměnná `Konec` má hodnotu `false`),
- inicializační funkce `Init()` otevře všechny potřebné soubory, inicializuje zásobník (vloží symboly # a S, resp. symboly `S_HASH` a `S_NS`) a provede první volání funkce lexikálního analyzátoru, ukončující funkce `Done()` zase vše uklidí, bude volána po ukončení celé analýzy,
- funkce `vystup()` přepíše svůj parametr na výstup.

Definice funkcí pracujících se samotným zásobníkem zde neuvádíme, jejich implementace závisí na typu použitého zásobníku (statický nebo dynamický), a taky může být hodně odlišná v různých programovacích jazycích.

Postup implementace si ukážeme na následujícím příkladu.

Příklad

Naprogramujeme syntaktickou analýzu podle gramatiky z příkladu na straně 68 s těmito pravidly a rozkladovou tabulkou:

$S \to AB$ ①
$A \to CD$ ②
$B \to +AB \mid -AB \mid \varepsilon$ ③, ④, ⑤
$C \to (S) \mid i \mid n$ ⑥, ⑦, ⑧
$D \to *CD \mid /CD \mid \varepsilon$ ⑨, ⑩, ⑪

	i	n	$+$	$-$	$*$	$/$	$($	$)$	\$
S	e1	e1					e1		
A	e2	e2					e2		
B			e3	e4				e5	e5
C	e7	e8					e6		
D			e11	e11	e9	e10		e11	e11

Využijeme datové typy a proměnné, které jsme si zavedli už u lexikální analýzy. Také si vytvoříme inicializační a úklidovou funkci.

```
// výčtový typ pro typy symbolů:
enum TTypSymbolu { S_NOTHING, S_ENDOFFILE,
  S_LPAR, S_RPAR, S_ID, S_NUM,
  S_PLUS, S_MINUS, S_MUL, S_DIV,
  S_NS, S_NA, S_NB, S_NC, S_ND, S_HASH }; // symboly pro neterminály a konec
    zásobníku

struct TSymbol {
  TTypSymbolu typ;          // identifikace symbolu, jeho typ
  string atrib;             // sémantický atribut symbolu
};

struct TVstup {
  char znak;                // zpracovávaný řádek
  int cisloRad;             // číslo řádku ve vstupním souboru
  int pozice;               // pozice posledního načteného znaku na řádku
  bool konec;               // lexikální analyzátor indikuje, že celý vstup byl načten
};

TSymbol symbol;              // právě zpracovávaný symbol načítaný lex. analyzátorem
string nazevsouboru;         // název zdrojového (vstupního) souboru
TVstup vstup;                // proměnná pro právě načtený znak ze vstupního souboru
...
bool konec;                  // proměnná určující konec syntaktické analýzy

TTypSymbolu vrchol_zas;      // symbol na vrcholu zásobníku
TZasobnik zasobnik;          // zásobník, prvky jsou typu TTypSymbolu
```

```
// funkce pro inicializaci překladu, může inicializovat taky lexikální analýzu
void init() {
  ...                              // inicializace vstupu a výstupu, lex. analýzy
  konec = false;
  vytvor_zasobnik();
  pridej_do_zasobniku(S_HASH);     // symbol konce zásobníku
  pridej_do_zasobniku(S_NS);       // startovací symbol gramatiky
  lex();                           // načte první symbol ze vstupu do proměnné symbol
}

// úklidová funkce, zavolá se po ukončení překladu
void done() {
  zlikviduj_zasobnik();            // uvolní paměť zabranou zásobníkem
  ...                              // uzavření vstupu a výstupu
}
```

V rozkladové tabulce jsou operace expanze, pop a akceptování vstupu, pro které si naprogramujeme funkce, a také funkci `error` pro chybová hlášení (v tabulce odpovídá prázdným buňkám).

```
void expand(int cislo_pravidla) {
  switch (cislo_pravidla) {
    case 1: pridej_do_zasobniku(S_NB);            // S -> AB
            pridej_do_zasobniku(S_NA);
            break;
    case 2: pridej_do_zasobniku(S_ND);            // A -> CD
            pridej_do_zasobniku(S_NC);
            break;
    case 3: pridej_do_zasobniku(S_NB);            // B -> +AB
            pridej_do_zasobniku(S_NA);
            pridej_do_zasobniku(S_PLUS);
            break;
    case 4: pridej_do_zasobniku(S_NB);            // S -> -AB
            pridej_do_zasobniku(S_NA);
            pridej_do_zasobniku(S_MINUS);
            break;
    ...
    }
  vystup(cislo_pravidla);           // zápis čísla použitého pravidla na výstup
}

// chyba syntaxe; vypíše číslo řádku, pozici na řádku a řetězec s daným hlášením:
void error(const string& hlaska) {
  konec = true;
  cout << "Chyba při syntaktické analýze na řádku " << vstup.cisloRad << ", pozici "
    << vstup.pozice << ": " << hlaska;
}

// volá se v případě, že na vstupu a na zásobníku je tentýž terminál:
void pop() {
  if (symbol.typ == vrchol_zasobniku) lex();   // lex. analyzátor načte další symbol
  else error("chybný symbol na vstupu - "+vypisTyp(symbol.typ));
}

// volá se v případě, že celý vstup je zpracovaný a celý zásobník také:
void accept() {
  konec = true;
}
```

Zde ve výpisech předpokládáme použití knihoven `iostream` a `string`, ale jsou samozřejmě i jiné možnosti. Také můžeme zvážit použití výjimek.

Dále potřebujeme „řídicí“ funkci, kterou jsme pojmenovali `akce()`. Tato funkce plní roli toho, kdo prochází tabulkou a rozhoduje (podle určeného řádku a sloupce), kterou funkci (pro expanzi,

pop, akceptování nebo hlášení chyby) je třeba v daném kroku zavolat. Schematicky je rozhodování o konkrétní buňce tabulky následující:

```
switch (řádek) {
  case první_řádek: switch (sloupec) {
                case první_sloupec: obsah_buňky_[1,1]; break;
                case druhý_sloupec: obsah_buňky_[1,2]; break;
                ...
                default: error(...);
              } break;
  case druhý_řádek: switch (sloupec) {
                case první_sloupec: obsah_buňky_[2,1]; break;
                case druhý_sloupec: obsah_buňky_[2,2]; break;
                ...
                default: error(...);
              } break;
  ...
  case terminál: pop; break;
  case dno_zásobníku: if (konec_vstupu) accept(); else error(...);
}
```

Vnější switch rozhoduje mezi řádky, vnitřní switche rozhodují mezi sloupci v rámci řádku. Podle našeho příkladu to bude takto:

```
void akce() {
  vrchol_zasobniku = vyjmi_ze_zasobniku();
  switch (vrchol_zasobniku) {
    case S_NS: switch(symbol.typ) {   // pro řádek tabulky s neterminálem S
        case S_ID:
        case S_NUM:
        case S_LPAR: expand(1); break;
        default: error("chybný symbol na vstupu - "+vypisTyp(symbol.typ));
      } break;

    case S_NA: switch(symbol.typ) {   // pro řádek tabulky s neterminálem A
        case S_ID:
        case S_NUM:
        case S_LPAR: expand(2); break;
        default: error("chybný symbol na vstupu - "+vypisTyp(symbol.typ));
      } break;

    case S_NB: switch(symbol.typ) {   // pro řádek tabulky s neterminálem B
        case S_PLUS: expand(3); break;
        case S_MINUS: expand(4); break;
        case S_RPAR:
        case S_ENDOFFILE: expand(5); break;
        default: error("chybný symbol na vstupu - "+vypisTyp(symbol.typ));
      } break;

    case S_NC: switch(symbol.typ) {   // pro řádek tabulky s neterminálem C
        case S_ID: expand(7); break;
        case S_NUM: expand(8); break;
        case S_LPAR: expand(6); break;
        default: error("chybný symbol na vstupu - "+vypisTyp(symbol.typ));
      } break;

    case S_ND: switch(symbol.typ) {   // pro řádek tabulky s neterminálem D
        case S_MUL: expand(9); break;
        case S_DIV: expand(10); break;
        case S_PLUS:
        case S_MINUS:
        case S_RPAR:
```

```
        case S_ENDOFFILE: expand(11); break;
        default: error("chybný symbol na vstupu - "+vypisTyp(symbol.typ));
      } break;

    case S_ID:          // pro řádky tabulky s terminály
    case S_NUM:
    case S_PLUS:
    case S_MINUS:
    case S_MUL:
    case S_DIV:
    case S_LPAR:
    case S_RPAR: pop(); break;
    case S_HASH:       // pro poslední řádek tabulky se symbolem konce zásobníku
        if (symbol.typ == S_ENDOFFILE) accept();
        else error("chybný symbol na vstupu - "+vypisTyp(symbol.typ));
        break;

    default: error("chybný symbol na vstupu - "+vypisTyp(symbol.typ));
  } // konec hlavního switche
}
```

Celá syntaktická analýza (a s ní simultánně i lexikální) proběhne po volání hlavní funkce syntaktické analýzy pojmenované `S_analyza()`.

```
void S_analyza() {
  init();
  while(!konec)
    akce();
  done();
}
```

Implementace metodou přepisu rozkladové tabulky má své výhody i nevýhody. Podívejme se nejdřív na výhody:

- překladač bude fungovat podobným způsobem, jakým se používá „papírová“ rozkladová tabulka, nemusíme vymýšlet nic nového,
- nepoužíváme rekurzi (neřešíme problém hloubky rekurze s prostorovou složitostí), přesněji – místo rekurze s potenciálním rizikem zacyklení a tedy nekonečného průchodu používáme zásobník, u kterého je taktéž riziko přetečení paměti, pokud je chyba v návrhu struktury jazyka.

Nevýhody metody:

- u překladů zahrnujících např. matematické výrazy se hůře implementuje sémantika, případně je třeba pro matematické výrazy se sémantikou využít jinou metodu, k čemuž se dostaneme později,
- potřebujeme naprogramovaný zásobník.

3.6.5 Implementace metodou rekurzivního sestupu

U metody rekurzivního sestupu nepotřebujeme žádný zásobník, používáme klasickou rekurzi[2]. Nemusíme dělat rozkladovou tabulku, ale potřebujeme všechny množiny, které bychom pro konstrukci rozkladové tabulky použili.

Pro každé pravidlo $A \to \alpha$ vytvoříme množinu signatur, jejíž vzorec ve skutečnosti už dobře známe:

$$\mathrm{FS}(A, \alpha) = \mathrm{FIRST}(\alpha \cdot \mathrm{FOLLOW}(A)) \tag{3.23}$$

[2] Ve skutečnosti zásobník používáme, ale jde o zásobník pro uživatelský režim běhu programu, který má každý program včetně překladačů pro rekurzivní volání funkcí a ukládání jejich parametrů a lokálních proměnných.

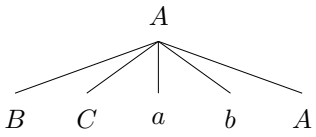
Tyto množiny pak použijeme pro testování při rozhodování mezi pravidly přepisujícími tentýž neterminál. U předchozí metody jsme jako základ pro programování použili rozkladovou tabulku a v zásobníku jsme simulovali derivaci podle konkrétního vstupu, zde pomocí pravidel budeme přímo skládat derivační strom.

Podle neterminálů vytvoříme ekvivalentně pojmenované funkce – třeba pro neterminál V vytvoříme funkci `V()` nebo `NV()` (tak, abychom se v kódu vyznali). Pro terminály (všechny) budeme mít jedinou funkci, kterou nazveme `pop()`, aby se nám to oproti předchozí metodě nepletlo, protože tato bude mít podobnou roli (třebaže se zásobníkem pracovat nebudeme): porovná terminál z pravidla s terminálem na vstupu, a když jsou stejné, načte další symbol ze vstupu (zavolá funkci `lex()`).

Příklad

Jak víme, ve vnitřních uzlech derivačního stromu se nacházejí neterminály, a tedy přítomnost na konkrétní pozici derivačního stromu pro nás bude znamenat, že spustíme funkci odpovídající neterminálu v daném uzlu derivačního stromu.

Předpokládejme, že už víme, že v daném uzlu derivačního stromu máme zpracovat pravidlo $A \to BCabA$. Tento uzel je tedy označen neterminálem A. Dále budeme konstruovat jeho podstrom, nejdřív nejbližší potomky podle pravé strany pravidla.



Při „ručním“ používání by to znamenalo, že vytvoříme potomky uzlu A, kteří budou ohodnoceni postupně symboly B, C, a, b, A, jak vidíme na obrázku vpravo. Pak přejdeme k symbolu B a podle vhodného pravidla vytvoříme jeho potomky...

Při programování postupně zavoláme funkce pro neterminály B, C, pak funkci `pop()` pro terminály a, b, a dále funkci pro neterminál A. Kód tedy bude vypadat takto:

```
void A() {
  ... // tady bude výběr pravidla, pravděpodobně s využitím switche
  // větev pro pravidlo A -> BCabA:
      B();
      C();
      pop(a);
      pop(b);
      A();
  ...
}
```

Takže zatímco bychom „na papíře“ kreslili čáry a zapisovali potomky, tady voláme odpovídající funkce. V každé funkci potřebujeme nejdřív rozhodnout, které pravidlo bude využito pro další patro stromu, k čemuž využijeme zmíněné množiny $\mathrm{FS}(A, \alpha)$. Pokud se dostaneme do větve pro pravidlo $A \to BCabA$, zavoláme funkce podle symbolů odpovídajících $BCabA$. Uvnitř funkce B bude opět členění podle pravidel, v části pro dané pravidlo spustíme funkce odpovídající tomu, co je na pravé straně pravidla pro B, atd.

Pokud je rekurze v pravidlech gramatiky, bude rekurze i v odpovídajících funkcích. Zde například vidíme přímou rekurzi neterminálu A, což se projevuje v tom, že ve funkci `A()` voláme rekurzívně funkci `A()`.

Je na nás, jak si nazveme funkce odpovídající neterminálům. Určitě bychom v názvu měli mít označení daného neterminálu, ať se nám to při programování neplete. Zde vidíme jednoznakové názvy funkcí, ale místo `A()` to může být třeba `NA()` nebo `symA()`.

Funkce, kterou jsme zde pro přehlednost nazvali `pop()`, se v literatuře často nazývá `expect()`.

Shrňme si tedy postup implementace metodou rekurzivního sestupu:

1. Vytvoříme funkce pro inicializaci `Init()` a úklid `Done()` (budou podobné jako při implementaci přepisem rozkladové tabulky, jen tam nebude zásobník). V inicializační funkci zavoláme funkci `Lex()`, čímž přednačteme jeden symbol.
2. Podle neterminálů vytvoříme ekvivalentně pojmenované funkce – třeba pro neterminál V vytvoříme funkci `V()`. V těle každé funkce bude switch podle pravidel přepisujících daný neterminál (stejně jako u předchozí metody – vybírá se podle symbolu na vstupu), ve větvi pro konkrétní pravidlo budou postupně volány funkce odpovídající symbolům z pravé strany pravidla.
3. Protože v kořeni stromu je startovací symbol gramatiky (obvykle S, ale může se samozřejmě jmenovat jinak), celá syntaktická analýza začne právě voláním funkce odpovídající tomuto symbolu. Vše další se už zavolá podle toho, jak jsou vybírána pravidla pro konkrétní neterminály v jednotlivých volaných funkcích.
4. Rekurzívní volání probíhá zleva doprava a shora dolů, tedy i vstup je čten zleva doprava.
5. Když skončí všechny rekurzivní výpočty pro jednotlivé větve a vstup je celý přečtený (na vstupu je \$, resp. `S_ENDOFFILE`), akceptujeme vstup.

Úkol

Vlastně se jedná o grafový algoritmus. Strom je ve skutečnosti druhem grafu (to víme z definice), a zde konstruujeme (resp. prohledáváme) graf metodou do hloubky, tedy postupně po jednotlivých větvích zleva. Pokud nevíte, jak probíhá prohledávání stromu do hloubky, pak si ten postup najděte a nastudujte.

Budeme potřebovat tyto funkce:

- `Init()` pro inicializaci výpočtu, `Done()` pro ukončení,
- `S()`, `A()`, `B()`, ... – pro každý neterminál vytvoříme stejně nazvanou funkci, resp. vhodně je nazveme podle toho, jak máme nazvané neterminály (neterminál nemusí nutně být jedno písmeno) a tak, abychom se v kódu vyznali,
- `pop()` ověří shodnost terminálu na vstupu se symbolem, který je parametrem této funkce, pak načte další symbol ze vstupu zavoláním funkce `lex()`, programujeme podobně jako `pop()` z předchozí metody,
- `error()` ošetří chybu, která se vyskytla při překladu,
- můžeme mít také zastřešující funkci `S_analyza()` jako u předchozí metody.

Příklad

Opět použijeme gramatiku z příkladu na straně 68 s těmito pravidly:

$S \to AB$	①
$A \to CD$	②
$B \to +AB \mid -AB \mid \varepsilon$	③, ④, ⑤
$C \to (S) \mid i \mid n$	⑥, ⑦, ⑧
$D \to *CD \mid /CD \mid \varepsilon$	⑨, ⑩, ⑪

Vytvoříme porovnávací množiny FS pro jednotlivá pravidla:

$\mathrm{FS}(S, AB) = \{(, i, n\}$ $\mathrm{FS}(B, \varepsilon) = \{\$,)\}$ $\mathrm{FS}(D, *CD) = \{*\}$
$\mathrm{FS}(A, CD) = \{(, i, n\}$ $\mathrm{FS}(C, (S)) = \{(\}$ $\mathrm{FS}(D, /CD) = \{/\}$
$\mathrm{FS}(B, +AB) = \{+\}$ $\mathrm{FS}(C, i) = \{i\}$ $\mathrm{FS}(D, \varepsilon) = \{+, -, \$,)\}$
$\mathrm{FS}(B, -AB) = \{-\}$ $\mathrm{FS}(C, n) = \{n\}$

Použijeme podobné datové typy jako v předchozí metodě, v seznamu symbolů však nebudeme potřebovat symboly pro neterminály (protože zde jsou reprezentovány funkcemi, nikoliv prvky výčtového typu). V inicializační funkci nesmíme zapomenout na zavolání funkce `lex()`, protože i zde potřebujeme jeden přednačtený symbol ze vstupu. Naopak rozhodně zapomeneme na funkce pracující se zásobníkem.

```
// výčtový typ pro typy symbolů:
enum TTypSymbolu { S_NOTHING, S_ENDOFFILE, S_LPAR, S_RPAR, S_ID, S_NUM,
  S_PLUS, S_MINUS, S_MUL, S_DIV };

struct TSymbol {
  TTypSymbolu typ;          // identifikace symbolu, jeho typ
  string atrib;             // sémantický atribut symbolu
};

struct TVstup {
  char znak;                // zpracovávaný řádek
  int cisloRad;             // číslo řádku ve vstupním souboru
  int pozice;               // pozice posledního načteného znaku na řádku
  bool konec;               // lexikální analyzátor indikuje, že celý vstup byl načten
};

TSymbol symbol;              // právě zpracovávaný symbol načítaný lex. analyzátorem
string nazevsouboru;         // název zdrojového (vstupního) souboru
TVstup vstup;                // proměnná pro právě načtený znak ze vstupního souboru
...
bool konec;                  // proměnná určující konec syntaktické analýzy

// funkce pro inicializaci překladu, může inicializovat taky lexikální analýzu
void init() {
  ...                             // inicializace vstupu a výstupu, lex. analýzy
  konec = false;
  lex();                          // načte první symbol ze vstupu do proměnné symbol
}

// úklidová funkce, zavolá se po ukončení překladu
void done() {
  ...                             // uzavření vstupu a výstupu
}
```

Hlavní funkce bude velmi jednoduchá:

```
void S_analyza() {
  init();
  S();      // zde spustíme syntaktickou analýzu, tímto začíná derivační strom
  done();
}
```

Dále potřebujeme funkce reprezentující symboly. Nejdřív naprogramujeme funkci pro zpracování terminálů, bude velmi podobná stejně nazvané funkci v předchozí metodě:

```
void pop(TTypSymbolu terminalZPravidla) {
  // porovnáme symbol ze vstupu se symbolem nalezeným v pravidle:
  if (symbol.typ == terminalZPravidla)
    lex();
  else error("chybný symbol na vstupu - "+vypisTyp(symbol.typ));
```

A teď k funkcím pro neterminály:

```
void S() {
  switch(symbol.typ) {
    case S_ID:          // pravidlo S -> AB
    case S_NUM:
    case S_LPAR:
      A();
      B();
      break;
    default: error("chybný symbol na vstupu - "+vypisTyp(symbol.typ));
  }
}

void A() {
  switch(symbol.typ) {
    case S_ID:          // pravidlo A -> CD
    case S_NUM:
    case S_LPAR:
      C();
      D();
      break;
    default: error("chybný symbol na vstupu - "+vypisTyp(symbol.typ));
  }
}

void B() {
  switch(symbol.typ) {
    case S_PLUS:        // pravidlo B -> +AB
      pop(S_PLUS);
      A();
      B();
      break;
    case S_MINUS:       // pravidlo B -> -AB
      pop(S_MINUS);
      A();
      B();
      break;
    case S_RPAR:        // pravidlo B -> epsilon
    case S_ENDOFFILE:
      break;
    default: error("chybný symbol na vstupu - "+vypisTyp(symbol.typ));
  }
}

void C() {
  switch(symbol.typ) {
    case S_LPAR:        // pravidlo C -> (S)
      pop(S_LPAR);
      S();
      pop(R_PAR);
      break;
    case S_ID:          // pravidlo C -> i
      pop(S_ID);
      break;
    case S_NUM:         // pravidlo C -> n
      pop(S_NUM);
      break;
    default: error("chybný symbol na vstupu - "+vypisTyp(symbol.typ));
  }
}

void D() {
  switch(symbol.typ) {
```

```
    case S_MUL:          // pravidlo D -> *CD
      pop(S_MUL);
      C();
      D();
      break;
    case S_DIV:          // pravidlo D -> /CD
      pop(S_DIV);
      C();
      D();
      break;
    case S_PLUS:         // pravidlo D -> epsilon
    case S_MINUS:
    case S_RPAR:
    case S_ENDOFFILE:
      break;
    default: error("chybný symbol na vstupu - "+vypisTyp(symbol.typ));
  }
}
```

Všimněte si, že součástí kódu jsou i větve odpovídající epsilonovým pravidlům. Tyto větve jsou (zatím) prázdné, přesto na ně nesmíme zapomenout, protože jinak by tyto případy „spadly“ do větví `default` a překladač by hlásil chybu i u správných vstupů. Až budeme přidávat sémantiku, tyto větve už nebudou prázdné.

Nyní se podívejme na výhody metody rekurzivního sestupu oproti metodě přepisu rozkladové tabulky:

- není nutné vytvářet rozkladovou tabulku, třebaže množiny signatur pro rozhodování mezi pravidly přepisujícími tentýž neterminál vytvořit musíme,
- nemusíme programovat zásobník, rekurze probíhá s použitím systémového zásobníku (k tomu se dostaneme, až budeme probírat implementaci funkcí v programovacím jazyce),
- sémantickou analýzu můžeme doplňovat přímo do tohoto kódu, u rekurzivních jazyků (včetně těch obsahujících běžné matematické výrazy) nepotřebujeme kombinaci s jinou metodou.

Nevýhody metody:

- je vhodné hlídat si hloubku rekurze, na což se podíváme při probírání sémantiky.

Úkoly

1. S následující gramatikou jsme se setkali v úkolu na straně 69, kde bylo naším úkolem sestrojit rozkladovou tabulku.

   $G = (\{S, A, B\},\ \{a, b, c, d, m\},\ P,\ S)$ s pravidly
   $S \to aAB \mid bBS \mid \varepsilon$
   $A \to cBS \mid \varepsilon$
   $B \to dB \mid m$

   Naprogramujte syntaktickou analýzu s použitím této tabulky, a to metodou přepisu rozkladové tabulky i metodou rekurzivního sestupu. Srovnejte oba způsoby implementace a všimněte si, na kterých místech se rozhoduje mezi pravidly přepisujícími tentýž neterminál a jak se v těchto metodách reaguje při zpracování neterminálu.

2. Vytvořte si vlastní jednoduchý jazyk obsahující rekurzivní matematické výrazy (běžné operátory, závorky s možností vnoření podvýrazů) nebo jejich ekvivalent (logické výrazy apod.). Sestrojte

gramatiku jazyka, a to tak, aby byla $LL(1)$, nebo proveďte transformaci na $LL(1)$ gramatiku. Zvolte si způsob implementace a naprogramujte syntaktický analyzátor.

3. Jazyk z předchozího příkladu obohaťte o příkazy. Jedním z příkazů by měl být přiřazovací příkaz

   $i = V$ (neterminál V představuje matematický výraz)

   Z dalších příkazů například příkaz pro načtení vstupu od uživatele do proměnné a příkaz pro výstup výsledku matematického výrazu, podle vlastních preferencí přidejte další příkazy.

   Pro takto upravený jazyk sestrojte rozkladovou tabulku nebo alespoň porovnávací množiny pro jednotlivá pravidla, vyberte si vhodnou metodu pro implementaci a naprogramujte syntaktickou analýzu.

3.7 Silné LL(k) gramatiky

3.7.1 Překladový automat pro silnou LL(k) gramatiku

Přílohy