Šárka Vavrečková

PROGRAMOVÁNÍ PŘEKLADAČŮ

Slezská univerzita v Opavě
Filozoficko-přírodovědecká fakulta
Ústav informatiky

Opava 2008

Anotace: Tato skripta jsou určena pro studenty předmětu Překladače. Čtenář po prostudování dokáže naprogramovat vlastní jednoduchý překladač. V jednotlivých kapitolách probereme základní fáze překladu, u každé návrh struktury fáze na teoretické úrovni a metody pro její naprogramování. V přílohách jsou tři souhrnné příklady zahrnující jak návrh struktury překladače, tak i jeho naprogramování.

Programování překladačů

Šárka Vavrečková

Lektoři: Doc. RNDr. Alica Kelemenová, CSc.
Doc. RNDr. Petr Šaloun, Ph.D.

Vydavatel: Slezská univerzita v Opavě
Filozoficko-přírodovědecká fakulta
Bezručovo nám. 13, 746 01 Opava

Tisk: Ediční středisko FPF SU v Opavě, 2008

ISBN: 978-80-7248-493-5

Sázeno v systému LaTeX

Obsah

Seznam obrázků

Seznam tabulek

Seznam definic

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*, 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 PASCALu a DELPHI. Je to především proto, že syntaxe jazyka PASCAL je všeobecně známá, ovládá

ji každý student, PASCAL má velmi dobrou podporu práce s množinami a také je vhodnější z didaktických důvodů. 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.

V Opavě dne 2. července 2008

Šárka Vavrečková

KAPITOLA 1

Překladač a jeho struktura

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 dále podrobně rozváděna v následujících kapitolách zabývajících se jednotlivými částmi překladu.

1.1 Základní pojmy

Definice 1.1 (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.*

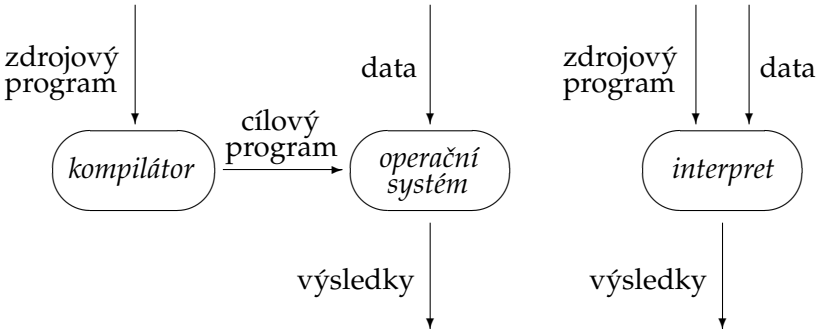
Podle typu cílového programu rozlišujeme tyto druhy překladačů:

kompilátor (generační překladač) je překladač, který má na vstupu program ve vyšším programovacím jazyce (FORTRAN, PASCAL, C, C++, DELPHI (vlastně 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), 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; typický příklad je JAVA (mezikód se zde nazývá bytecode) nebo .NET jazyky (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 v případě programu, který uchovává jeho uživatel, jde většinou o textový soubor, který obvykle zabírá mnohem méně místa než obdobný cílový program přeložený kompilátorem, a 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]). 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*[2]	
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, a tedy vlastně patří i ke kompilačním překladačům.

1.2 Hlavní části překladače

Překladač rozdělíme 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.

[1] Kompatibilita je bez problémů snad až na malé drobnosti, jako jsou různé způsoby označování konce řádku v textovém souboru: Windows používají pro „zařádkování“ dvojici znaků – znaky s ASCII kódy 13 a 10 (CRLF), Unixové systémy a Mac používají pouze jeden z nich (Unix 10 – LF, Mac 13 – CR).

[2] Pokud v případě interpretovaného programu překladač již spuštěn, může být u jednoduchých jazyků spuštění interpretovaného programu rychlejší.

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* zahrnuje lexikální, syntaktickou a sémantickou analýzu. Je do značné míry nezávislá na cílovém systému, generuje vnitřní formu programu.
- *Zadní část* 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 1.2 (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).*

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.

V tomtéž průchodu obvykle bývají 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, ...

Naproti tomu 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.

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 (nebo je nechává vkládat ze souboru) 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, 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 (editorem) 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 vždy (nebo alespoň téměř vždy) 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 zdrojový 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 nebo procedury – 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. Souvisí 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 (vytváří lexikální analyzátor) a YACC nebo BISON (syntaktický analyzátor)[3]. Jsou k dosažení jako freeware na Internetu, lze také zakoupit licence těchto programů v propracovanějších verzích.

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 (PDA, mobilní telefony) nebo roboty.

Dnes se běžně problémy překladu pro jinou platformu řeší použitím emulátorů hardwaru nebo operačního systému (emulátorům se věnujeme v předmětu *Operační systémy*). *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.

Při programování aplikací (včetně překladačů) pro mobilní telefony nebo PDA lze obvykle sehnat vývojová prostředí včetně emulátorů, v některých případech volně šiřitelná. Například Symbian OS SDK je balík programů pro vývojáře aplikací pro mobilní telefony s operačním systémem Symbian[4], a z nejnovějších existuje Android SDK[5]. Při programování některých mobilních telefonů s operačním systémem Symbian je také používáno vývojové prostředí Mophun[6] pracující na Windows i Linuxu, taktéž obsahující emulátor Symbianu.

Problémy s kompatibilitou částečně odpadají u hybridních jazyků (výsledný kód je sice binární, ale nezávislý na operačním systému), proto je při programování mobilních telefonů velmi oblíbený jazyk JAVA.

[3] Informace například na http://dinosaur.compilertools.net/.
[4] Volně dostupný na http://developer.symbian.com/main/tools/sdks/.
[5] Volně dostupný na http://code.google.com/android/download.html.
[6] Informace na http://mophun.com.

Při programování aplikací pro herní konzole je obvykle třeba používat kromě běžného počítače také herní konzoli, k ní dále hardwarové rozšíření (například pro Playstation je to buď PS2 Development Tool nebo Sony PS2 Linux Kit[7]) a případně vývojové prostředí. Existují také emulátory (například PCSX2 nebo nSX2 emulující Playstation 2[8], CxBx nebo Xeon emulující XBox[9]), avšak především vzhledem k značně odlišnému charakteru hardwaru konzolí (masivní paralelizace grafických operací, zvláště na PS2) jsou aplikace běžící v těchto emulátorech pomalejší než na skutečném hardwaru, což může ztěžovat programování.

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, narozdíl 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 portování. Linux dnes najdeme nejen na strojích kompatibilních s procesory Intel, ale také PowerPC, Alpha, Sparc, PDA, 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 spustitel-

[7] Informace na http://developer.symbian.com/main/tools/sdks/, http://www.playstation.com.
[8] Informace na http://www.pcsx2.net/ a http://nsx2.emulation64.com/index2.html.
[9] Informace na http://www.emulator-zone.com/doc.php/xbox/.

ného souboru v různých operačních systémech bez jakýchkoliv úprav[10]). Výjimkou jsou programy, ve kterých je závislost na hardwaru nebo operačním systému nutností (např. 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. Je to nejobvyklejší forma pro překladače pro OS MS-DOS (BASIC, PASCAL, C, TASM, NASM, . . .). Pod Windows se běžně používala pouze pro nejstarší překladače pracující přímo s API rozhraním[11], dnes se s určitým rozšířením (především pro větší komfort práce v prostředí) používá u editorů třetích stran – pro Windows[12] PSPad, Crimson Editor, CodePad, Notepad++, TED Notepad, atd., pro Linux vim, emacs, kate a další součásti standardních distribucí.

Grafický editor. Takovéto editory se používají pro velmi jednoduché programovací jazyky, případně hry nebo rozhraní pro vytváření her. Na obrázku 1.2 na straně 14 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 být reprezentovány názvem nebo jejich pořadovým číslem, obrázek, který může být animován, obsahuje ve skutečnosti několik obrázků, které se v krátkých intervalech střídají).

[10] 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`.

[11] API rozhraní je sada funkcí a objektů, které jsou přímo součástí operačního systému. Lze tak například vytvořit okno, zobrazit standardní dialog, zaokrouhlit číslo apod. API je celé uloženo v systémových dynamicky linkovaných knihovnách.

[12] Většinou jsou kromě svých domovských stránek dostupné na http://www.slunecnice.cz.

Obrázek 1.2: Ukázka prostředí grafického editoru určeného pro děti

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, jakou používá ve svých překladačích například Ing. Soukup[13]. Spočívá v postupném vytváření struktury podobné n-árnímu stromu, která se vyhodnocuje shora dolů a zleva doprava. S touto formou kódu se také můžeme setkat v modelovacím jazyce UML.

[13]SGP – Soukup Graphics Programming, informace na http://www.sgpsys.com/cz/. Známým programem je především *Baltazar*, jeho zjednodušenou verzi *Baltík* řadíme spíše do předchozí skupiny – grafických editorů.

Vývojové prostředí. Tyto „editory“ jsou nyní nejpoužívanější (VISUAL BASIC, 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 kódu[14]. 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“.

V Linuxu se také můžeme setkat s oddělením programování grafického rozhraní od zbytku aplikace. Příkladem aplikace specializované na vytvoření grafického prostředí je GLADE.

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 někteří programátoři malou nápovědu občas uvítají). 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[15]. 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.

[14] Editory podporující skrývání kódu umožňují „sbalit“ části kódu podobně jako položky adresáře v zobrazení stromové struktury adresářů (složek), a to jednoduše klepnutím na značku vlevo od řádku s kódem nadřízeným tomu skrývanému. Tato vlastnost má zvyšovat přehlednost kódu.

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

- *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 ke kapitole 1

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,
   - zda pro něj existuje možnost vygenerovat cílový kód a jakým způsobem se to provádí (případně zjistěte volně dostupné překladače[16]).

FLEX	LISP	PERL	SQUEAK
GOEDEL	LOGO	PROLOG	SMALLTALK
HASKELL	LUA	PYTHON	TCL
JAVA	MERCURY	RUBY	TCL/TK

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).

[16] Jedním z nejlepších zdrojů překladačů je například http://www.thefreecountry.com/, a samozřejmě http://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

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.

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

2.1 Popis lexikální struktury jazyka

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

Příklad 2.1

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. Pokud seznam symbolů uložíme do textového souboru, ukládáme symboly každý zvlášť na jednom řádku. Délka identifikátoru je konstantní, doplněná mezerami, pak následuje hodnota atributu (případně více atributů). 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):

```
TSymbol = record
  typ:   TTypSymbolu;          // identifikace symbolu
  atrib: string;               // atribut symbolu
end;
```

Při stanovení lexikální analýzy 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 2.2

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

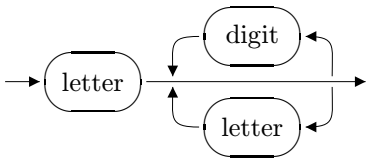
- celá nezáporná čísla (pro konstanty),
- rezervované identifikátory – klíčová slova: `BEGIN`, `END`, `VAR`, `CONST`, `IF`, `THEN`, `ELSE`, `PRINT`,

- 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.

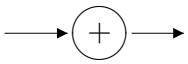
Příklad 2.3

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 2.2. Klíčová slova zatím nebudeme odlišovat od ostatních identifikátorů.

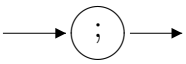


(a) Graf pro `S_ID`

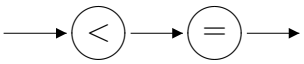
(b) Graf pro `S_NUM`



(c) Graf pro `S_PLUS`



(d) Graf pro `S_SEM`



(e) Graf pro `S_LQ`

Obrázek 2.1: Syntaktické grafy některých symbolů

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 2.1(a). 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 příkladu 2.2.
$G = (N, T, P, S)$, $T = \Sigma$ (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):

$S \rightarrow l \mid lA$ 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* | *konec_řádku*

2.2 Rozpoznávání symbolů

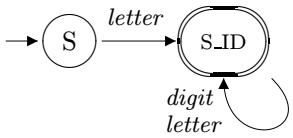
V předchozí kapitole jsme určili lexikální strukturu jazyka pomocí regulární gramatiky. Gramatika dokáže jazyk popsat, ale pokud chceme zjistit, zda zadané slovo patří do jazyka (tj. určit ze vstupního řetězce, o jaký symbol jde), potřebujeme konečný automat, který bude pracovat takto:

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.

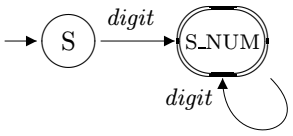
Příklad 2.4

Podle regulární gramatiky v příkladu 2.3 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 na straně 22. Ostatní diagramy jsou podobné, jejich vytvoření necháváme na čtenáři.

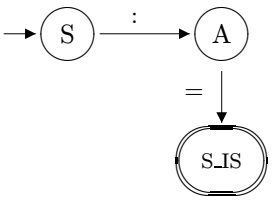
Všechny tyto stavové diagramy popisují konečné deterministické automaty, jejichž koncové stavy představují symboly. Když vytvoříme stavové diagramy pro všechny symboly a shrneme je (tj. sloučíme počáteční stavy S stavových diagramů pro všechna slova jazyka), získáme konečný deterministický automat rozpoznávající jazyk z příkladů 2.2–2.4.



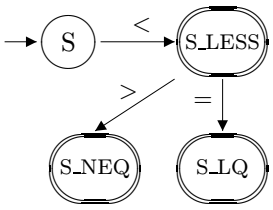
(a) Konečný automat pro `S_ID`



(b) Konečný automat pro `S_NUM`



(c) Konečný automat pro symbol „: =“



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

Obrázek 2.2: Konečný automat pro některé symboly

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 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. Programový kód v následujícím textu bude psán v PASCALu nebo odvozených jazycích, protože tento jazyk ovládá téměř každý (budoucí) informatik.

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 příkladů 2.2–2.4, 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.

```
type
  TTypSymbolu = (S_BEGIN, S_END, S_CONST, S_VAR, S_ID, S_NUM, S_LPAR, S_RPAR,
    S_SEM, S_IF, S_THEN, S_ELSE, S_PRINT, S_IS, S_PLUS, S_MINUS, S_MUL, S_DIV,
    S_EQ, S_NEQ, S_LESS, S_GRT, S_LQ, S_GQ);

  TSymbol = record
    typ: TTypSymbolu;        // identifikace (název) symbolu
    atrib: string;           // atributy
  end;
```

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, 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.

Předpokládejme, že zdrojový program je ve formě textového souboru. Protože přístupy na paměťové médium jsou časově náročné, budeme načítat text ne po jednotlivých znacích, ale *po řádcích*. Po vyhodnocení celého řádku načteme následující řádek a tak postupujeme až ke konci zdrojového souboru. Pro uschování načtené části vstupu zvolíme tento záznam:

```
type
  TZnak = record
    rad:    string;     // zpracovávaný řádek
    pozice: byte;       // pozice posledního načteného znaku na řádku
    delka:  byte;       // délka tohoto řádku
    cislo:  word;       // číslo řádku
  end;
```

„Aktivní“ – právě zpracovávaný – znak je ve vnitřní proměnné `rad[pozice]`. Procedura načte další řádek teprve tehdy, až zjistí, že došla na konec dříve načteného řádku. K tomu nám v záznamu slouží proměnné `pozice` a `delka`.

Proměnnou `cislo` 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 `pozice`.

Na konci souboru vrací procedura v proměnné `Znak.rad[Znak.pozice]` znak s kódem 0. Na konec každého řádku přidává mezeru, která nahrazuje znak konce řádku.

Pro jednoduchost náš překladač nebude rozlišovat velká a malá písmena, proto do procedury zahrneme převod malých písmen na velká.

```
var
  zdroj: Text;         // zdrojový program (textový soubor)
  symbol: TSymbol;     // proměnná pro zachycení načítaného symbolu
  znak: TZnak;         // proměnná pro uložení části vstupního souboru

procedure DejZnak;     // zajistí posun na další znak ze vstupu
var i: byte;
begin
  if eof(zdroj) then
    znak.rad[pozice] := #0             // zastupuje symbol konce souboru
  else with Znak do begin
    if delka = pozice then begin       // je nutné načíst další řádek
      readln(zdroj, rad);
      rad := rad + ' ';
      delka  := length(rad);
      pozice := 1;
      for i := 1 to delka do
        rad [i] := UpCase(rad [i]);    // převod na velká písmena
    end else inc(pozice);              // ještě nejsme na konci řádku
  end;
end;
```

Pro skutečný zdrojový jazyk bude implementace složitější, například procedura by měla automaticky vynechávat komentáře (ale přesto je zahrnovat do počtu řádků).

2.3.2 Metody pro konečné a nekonečné jazyky

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:

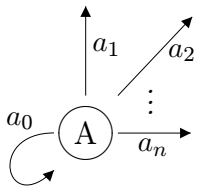
- implementace vhodné zejména pro nekonečné jazyky, které obsahují symboly s atributy (čísla, proměnné, ...), ostatní symboly mají ve zdrojovém textu jen krátké, několikaznakové vyjádření (operátory apod.),
- implementace vhodné především pro konečné jazyky obsahující symboly reprezentované ve zdrojovém programu delšími řetězci (klíčová slova).

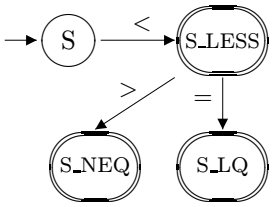
Ukazuje se, že výhodou může být zkombinování obou typů implementací, a to tak, že nejdříve načteme symbol metodou z první skupiny (zatím nerozlišujeme mezi klíčovými slovy a jinými identifikátory, vše, co začíná písmenem, bereme jako identifikátor), a pokud je to identifikátor, použijeme některou z metod druhé skupiny na odlišení klíčových slov a případně dalších speciálních identifikátorů.

A) Přímé stavové programování

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

```
while zn = a0 do begin
  DejZnak;
  zn := znak.rad[znak.pozice];
end;
case zn of
  a1: begin ... end;
  a2: begin ... end;
  ...
  else ...;
end; ...
```





Například podle obrázku 2.2(d) na straně 22 (je uveden také nad tímto textem vpravo) postupujeme následovně:

```
case znak.rad[znak.pozice] of        // jsme ve stavu S
  ...
  '<': begin                         // jsme ve stavu S_LESS
    DejZnak(znak);                   // posun ve vstupu na další znak
    case znak.rad[pozice] of
      '>': symbol.typ := S_NEQ;      // jsme ve stavu S_NEQ
      '=': symbol.typ := S_LQ;       // jsme ve stavu S_LQ
      else symbol.typ := S_LESS;     // zůstáváme ve stavu S_LESS
    end;
    ...                          // jiný typ symbolu
  end;
  else ...;                      // ošetření chyby
end;
```

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 2.5

Je dán konečný jazyk L = {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 = $\{i, f, t, h, e, n, l, s\}$

$S \to iA_1$	$S \to tA_2$		$S \to eA_5$
$A_1 \to f$	$A_2 \to hA_3$		$A_5 \to lA_6$
$[\Rightarrow IF]$	$A_3 \to eA_4 \mid iA_8$		$A_6 \to sA_7$
	$A_4 \to n$	$A_8 \to s$	$A_7 \to e$
	$[\Rightarrow THEN]$	$[\Rightarrow THIS]$	$[\Rightarrow ELSE]$

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

9 . . . chybový stav	10 . . . načteno *if*	12 . . . načteno *else*
	11 . . . načteno *then*	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, . . . , 8.

	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

```
const
  k_chyba = 9;      // chybový stav a koncové stavy
  k_if    = 10;
  k_then  = 11;
  k_else  = 12;
  k_this  = 13;
  MaxStav =  8;     // maximální číslo stavu
  MaxZnak =  8;     // maximální číslo znaku

var
  tab: array [0..MaxStav, 1..MaxZnak] of byte;
  symbol: TSymbol;

function DalsiZnak: byte;   external;
// Přečte ze vstupu jeden znak, vrátí číslo z intervalu 1..MaxZnak
// pro jeden ze znaků i, f, ..., nebo číslo 0 pro chybný vstup.

procedure NactiTabulkuPrechodu;  // zavoláme na začátku celého překladu
var
  i, j: byte;
begin
  for i := 0 to MaxStav do
    for j := 1 to MaxZnak do tab[i,j] := k_chyba;
  tab[0,1] := 1;  tab[0,3] := 2;  tab[0,5] := 5;  tab[2,4] := 3;
  tab[3,1] := 8;  tab[3,5] := 4;  tab[5,7] := 6;  tab[6,8] := 7;
  tab[1,2] := k_if;               tab[4,6] := k_then;
  tab[7,5] := k_else;             tab[8,8] := k_this;
end;
```

```
procedure Lex;
var
  stav: byte;
  znak: byte;
begin
  stav := 0;           // nulování, můžeme provést také pro "symbol"
  znak := DalsiZnak;   // přednačteme jeden znak ze vstupu
  while (not konec_vstupu) and (stav < k_chyba) do begin
    stav := tab[stav, znak];
    ... // případné další zpracování znaku, například přidání k atributu symbolu
    znak := DalsiZnak;
  end;  // while
  case stav of
    k_if:    symbol.typ := S_IF;
    k_then:  symbol.typ := S_THEN;
    k_else:  symbol.typ := S_ELSE;
    k_this:  symbol.typ := S_THIS;
    else  writeln('Chyba při zpracování');  // včetně stavu k_chyba
  end;
end;
```

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 v příkladu 2.5 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 políčka 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.

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

Příklad 2.6

Automat z příkladu 2.5 přepíšeme takto:

```
const
  k_chyba =  9;   k_else  = 12;     // chybový stav a koncové stavy
  k_if    = 10;   k_this  = 13;
  k_then  = 11;

var
  symbol: TSymbol;

function DalsiZnak: char;   external;
// načte ze vstupního souboru jeden znak a vrátí převedený na velké písmeno

procedure Lex;
var
  stav: byte;
  znak: char;
begin
  stav := 0;           // nulování, můžeme provést také pro "symbol"
  znak := DalsiZnak;
  while (not konec_vstupu) and (stav < k_chyba) do begin
    case stav of
      0: case znak of
           'I': stav := 1;
           'T': stav := 2;
           'E': stav := 5;
           else stav := k_chyba;
         end;
      1: if znak = 'F' then stav := k_if   else stav := k_chyba;
      2: if znak = 'H' then stav := 3      else stav := k_chyba;
      3: case znak of
           'I': stav := 8;
           'E': stav := 4;
           else stav := k_chyba;
         end;
      4: if znak = 'N' then stav := k_then else stav := k_chyba;
      5: if znak = 'L' then stav := 6      else stav := k_chyba;
      6: if znak = 'S' then stav := 7      else stav := k_chyba;
      7: if znak = 'E' then stav := k_else else stav := k_chyba;
      8: if znak = 'S' then stav := k_this else stav := k_chyba;
      else stav := k_chyba;
    end; // case
    znak := DalsiZnak;
  end; // while

  case stav of
    k_if:    symbol.typ := S_IF;
    k_then:  symbol.typ := S_THEN;
    k_else:  symbol.typ := S_ELSE;
```

```
    k_this:  symbol.typ := S_THIS;
    else writeln('Chyba při zpracování');
  end;
end;
```

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. U metod ukázaných na příkladech 2.5 a 2.6 musíme ještě přidat test znaku následujícího za symbolem, který nás zavedl do některého koncového stavu.

Budeme dále pokračovat v příkladu z kapitoly 2.3.1. Sestavíme proceduru `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 proceduru `ZpracujID`, která načtený atribut dále zpracuje a určí, zda nejde o klíčové slovo. Na konci každého symbolu se procedura zastaví a ve vyhodnocení vstupu pokračuje, až když je znovu volána.

```
var
  znak:   TZnak;       // znak načtený ze souboru
  symbol: TSymbol;     // zde ukládáme načtený symbol
...
procedure Lex;   // načte jeden symbol do globální proměnné symbol
begin        // procedura DejZnak byla už volána, načtený znak je v záznamu znak
  while (znak.rad[znak.pozice] = ' ') do DejZnak;
  case znak.rad[znak.pozice] of
    'A'..'Z': begin              // identifikátor nebo klíčové slovo
      symbol.atrib := znak.rad[znak.pozice];
      DejZnak;
      while (znak.rad[znak.pozice] in ['A'..'Z', '0'..'9']) do begin
        symbol.atrib := symbol.atrib + znak.rad[znak.pozice];
        DejZnak;
      end;
      ZpracujID(symbol.atrib);
    end;
```

```
    '0'..'9': begin                 // číslo
      symbol.atrib := znak.rad[znak.pozice];
      DejZnak;
      while (znak.rad[znak.pozice] in ['0'..'9']) do begin
        symbol.atrib := symbol.atrib + znak.rad[znak.pozice];
        DejZnak;
      end;
      symbol.typ := S_NUM;
    end;

    '<': begin                      // symbol '<' nebo '<=' nebo '<>'
      DejZnak;
      case znak.rad[znak.pozice] of
        '>': begin
          DejZnak;
          symbol.typ := S_NEQ;      // <>
        end;

        '=': begin
          DejZnak;
          symbol.typ := S_LQ;       // <=
        end;
      else symbol.typ := S_LESS;    // <
    end;
    ...                             // Podobně všechny ostatní symboly
    else ...                        // Ošetření chyby
  end;
end;
```

Dále musíme odlišit klíčová slova od ostatních identifikátorů a výsledky uložit do výstupního souboru. Proceduru můžeme sestavit více způsoby. První způsob je vhodný nejvýše pro velmi jednoduchý programovací jazyk s několika klíčovými slovy, my tento způsob *nebudeme používat*:

```
procedure ZpracujID(s: string);
begin
  if      s = 'BEGIN' then symbol.typ := S_BEGIN
  else if s = 'END'   then symbol.typ := S_END
  else if s = 'CONST' then symbol.typ := S_CONST
  else if s = 'VAR'   then symbol.typ := S_VAR
  ...  // atd. pro všechna klíčová slova
  else begin
    symbol.typ := S_ID; // Není to klíčové slovo
    symbol.atrib := s;  // Tento řádek v našem případě není nutný
  end;
end;
```

Z hlediska překladu a výsledného kódu překladače (časové složitosti překladu) je optimálnější, u jazyků s rozsáhlejší „slovní zásobou“ velmi výrazně, jiná metoda. Napíšeme proceduru jako konečný automat podle druhé nebo třetí metody z kapitoly 2.3.2.

Příklad 2.7

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

$S \to bA_1$ $A_1 \to eA_2$ $A_2 \to gA_3$ $A_3 \to iA_4$ $A_4 \to n$

$S \to eA_5$ $A_5 \to nA_6$ $A_6 \to d$

$S \to cA_7$ $A_7 \to oA_8$ $A_8 \to nA_9$ $A_9 \to sA_{10}$ $A_{10} \to t$

$S \to vA_{11}$ $A_{11} \to aA_{12}$ $A_{12} \to r$

$S \to iA_{13}$ $A_{13} \to f$

$S \to tA_{14}$ $A_{14} \to hA_{15}$ $A_{15} \to eA_{16}$ $A_{16} \to n$

$A_5 \to lA_{17}$ $A_{17} \to sA_{18}$ $A_{18} \to e$

$S \to pA_{19}$ $A_{19} \to rA_{20}$ $A_{20} \to iA_{21}$ $A_{21} \to nA_{22}$ $A_{22} \to t$

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

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

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

```
const
  PocetZnaku = 17; // Počet znaků, ze kterých se skládají klíčová slova
var
  tab: array [0..22, 1..PocetZnaku] of byte;

procedure NactiTabulku;
var
  i, j: byte;
begin
  for i := 0 to 22 do
    for j := 1 to PocetZnaku do tab[i,j] := k_chybovy;
  tab[ 0, 1] :=  1;   tab[ 0, 2] :=  5;   tab[ 0, 4] := 13;
  tab[ 0, 7] :=  7;   tab[ 0,10] := 14;   tab[ 0,11] := 11;
  tab[ 0,17] := 19;   tab[ 1, 2] :=  2;   tab[ 2, 3] :=  3;
  tab[ 3, 4] :=  4;   tab[ 4, 5] := 24;   tab[ 5, 5] :=  6;
```

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

function DejCisloZnaku(zn: char): byte;
// Pokud zn nepatří do abecedy, nad kterou jsou vytvořena klíčová slova,
// funkce vrátí hodnotu 0. Jinak vrací index znaku.
const
  Index: string [PocetZnaku] = 'BEGINDCOSTVARFHLP';
var
  i, v: byte;
begin
  v := 0;             // číslo 0 náleží nedefinovanému znaku
  i := 1;
```

```
  while (i <= PocetZnaku) do begin
    if (zn = Index [i]) then begin
      v := i; break;      // nalezen index (číslo) znaku v seznamu
    end;
    inc(i);
  end;
  DejCisloZnaku := v;
end;

procedure ZpracujID(s: string);
var
  stav:   byte;         // aktuální stav automatu
  pozice: byte;         // pozice v testovaném řetězci s
  delka:  byte;         // délka řetězce s
  znak:   byte;         // číslo znaku podle seznamu znaků klíčových slov
begin
  stav   := 0;
  pozice := 1;
  delka  := length(s);
  while (pozice <= delka) and (stav < k_chyba) do begin
    znak := DejCisloZnaku(s[pozice]);
    if (znak = 0) then stav := chybovy
                  else stav := tab[stav,znak];
    inc(pozice);
  end;
  case stav of
    k_begin: symbol.typ := S_BEGIN;
    k_end:   symbol.typ := S_END;
    k_const: symbol.typ := S_CONST;
    k_var:   symbol.typ := S_VAR;
    k_if:    symbol.typ := S_IF;
    k_then:  symbol.typ := S_THEN;
    k_else:  symbol.typ := S_ELSE;
    k_print: symbol.typ := S_PRINT;
    else begin
      symbol.typ := S_ID;
      symbol.atrib := s;
    end;
  end;
end;

procedure InitLex;
// Tato procedura je volána pouze jednou za celý překlad
begin
  ...             // Otevření vstupního souboru pro čtení, zpřístupnění
                  // přes proměnnou zdroj (textový soubor).
  NactiTabulku;   // Načteme tabulku přechodů do proměnné tab.
  DejZnak;        // Načteme do proměnné znak první znak souboru,
                  // v proceduře NactiSymbol se s tím počítá
end;
```

První způsob implementace používající prosté porovnávání řetězců je určitě velmi jednoduchý, rychlý a intuitivní. Časová složitost výpočtu[1] je však (zejména pro jazyky s větším množstvím klíčových slov) podstatně vyšší, než je únosné. Důvodem je vícenásobné procházení testovaného řetězce – v nejhorším případě, tedy když nejde o klíčové slovo, je alespoň začátek řetězce procházen při každém uvedeném porovnávání. Proto má smysl takto postupovat pouze u jazyků, které mají velmi málo klíčových slov a jsou postaveny především na jiných typech symbolů.

U druhého způsobu je časová složitost obecně mnohem nižší (každý znak slova je 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, chybový stav). Můžeme samozřejmě postupovat také metodou pro konečné jazyky s nižší prostorovou složitostí, která je ukázaná na příkladu 2.6 na straně 29.

2.4 Datové typy konstantních hodnot

Do této chvíle jsme pracovali pouze s programovacími jazyky, které měly jediný datový typ – celé nezáporné číslo. V praxi se však používají jazyky přijímající obvykle celá čísla (bez znaménka nebo se znaménkem), reálná čísla, znaky, řetězce, pole, záznamy, pointery, výčtové typy atd. Lexikální analyzátor se obvykle takovými rozlišeními nemusí zabývat, pokud ovšem nejde o konstanty.

Čísla můžeme nechat v znakové podobě tak, jak byla ve zdrojovém textu, nebo je předat dál v binárním tvaru ve vhodné reprezentaci (pak pro atribut nepoužijeme řetězec, ale variantní záznam, příp. v C union, kde jednotlivé možnosti budou odpovídat zvolenému datovému typu konstanty). Tuto reprezentaci volíme podle toho, co nám nabízí programovací jazyk, ve kterém překladač píšeme, například u celých čísel máme obvykle na výběr mezi těmito možnostmi:

- celé číslo se znaménkem na 2 B (integer)[3], rozmezí $-32\,768 \ldots 32\,767$,
- celé číslo bez znaménka na 2 B (word), rozmezí $0 \ldots 65535$,
- celé číslo se znaménkem na 1 B (short), rozmezí $-128 \ldots 127$,
- celé číslo bez znaménka na 1 B (byte, char), rozmezí $0 \ldots 255$.

[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.

[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.

[3] Skutečné množství paměti pro integer závisí na operačním systému – 2 B platí pro 16-bitový OS, 32-bitové operační systémy (momentálně nejpoužívanější) používají 4 B, v 64-bitových systémech zabírá integer 8 B, a od toho se odvíjí také rozmezí hodnot.

Vzhledem k tomu, že znaménko „−“ můžeme chápat jako zvláštní symbol, volíme spíše datové typy, které znaménko nepoužívají, ale díky tomu na stejně velkém paměťovém místě nabízejí větší rozsah pro kladné číslo. Pro racionální čísla s plovoucí desetinnou čárkou můžeme volit vždy tentýž datový typ nebo rozhodovat obdobně jako u celých čísel.

Příklad 2.8

V takovém jazyce byl napsán úsek programu:

```
CONST
  a = 224;
  b = - 224;
  c = - 5;
  d = 10000;
...
  prom := 25 * b + 8224;
```

Vyskytuje se zde celkem šest celočíselných konstant, u kterých je nutné určit datový typ. Toto rozlišení může provádět sémantický analyzátor nebo je lze přenechat lexikálnímu. V lexikálním analyzátoru postupujeme takto:

1. načteme řetězec s číslicemi (nebo průběžně načítáme),
2. převedeme řetězec na číslo (vytvoříme „meziprodukt“ představující nejuniverzálnější reprezentaci – použijeme datový typ zabírající nejvíce místa v paměti); nemusíme používat funkce, které jsou součástí programovacího jazyka, ve kterém píšeme překladač, převody jsou poměrně snadné,
3. porovnáváme načtené číslo s mezními hodnotami a podle toho určujeme přesný datový typ,
4. pokud má lexikální analyzátor přístup k informaci, zda jde o kladné nebo záporné číslo, můžeme tento fakt zohlednit při výběru datového typu, ovšem to se týká spíše definice pojmenované konstanty než výskytu konstanty ve výrazu.

V našem případě tedy bude výsledek takový (jsou uvedeny pouze číselné symboly, nikoliv ostatní včetně symbolu pro znaménko „−“):

```
S_NUM_BYTE   224
S_NUM_BYTE   224
S_NUM_BYTE   5
S_NUM_WORD   10000
S_NUM_BYTE   25
S_NUM_WORD   8224
```

Reprezentace konstantního *řetězce* není problém, pouze vzhledem k optimalizaci prostorové složitosti volíme vhodnou délku řetězce. Řetězec je obvykle ohraničen speciálními

znaky (jednoduché nebo dvojité uvozovky), takže lexikální analyzátor po nalezení prvního takového znaku pokračuje v načítání, dokud nenajde druhý, uzavírací znak řetězce. Uvozovací znaky nejsou symboly, z hlediska překladače jde pouze o pomocné znaky, které mu říkají, kde řetězec začíná a kde končí.

Dále můžeme ošetřit případ, kdy uživatel velmi dlouhý řetězec rozdělí na více menších řetězců a každý umístí na nový řádek (to umožňuje například programovací jazyk C). Pokud konstantní řetězce ukládáme do dostatečně rozsáhlých hodnot symbolů (jestliže je výstupem dynamická struktura nebo soubor, lze délku hodnot typu řetězec určovat též dynamicky), můžeme všechny tyto konstantní řetězce spojit do jediného. Pokud programovací jazyk umožňuje sčítání řetězců, lze jednotlivé řetězce načíst zvlášť a spojit je explicitně operátorem sčítání (není to obvyklý postup) nebo se lexikální analyzátor nemusí vůbec namáhat řešením těchto situací a výsledkem je prostě posloupnost řetězců, kterou zpracuje syntaktický analyzátor.

U konstantních *polí* a *záznamů* záleží na zvolené vnitřní reprezentaci jazyka a předepsaném tvaru definice těchto konstant. Obvyklé je zadávat pole jako výčet prvků oddělených čárkou a záznam jako posloupnost vnitřních proměnných a jejich hodnot, takže lexikální analyzátor tyto konstanty jako celek nemusí zpracovávat a předává je dál v rozloženém tvaru.

Úkoly ke kapitole 2

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 24, možnost kombinace metod v podkapitole 2.3.3 od strany 30).

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)
   - $L_6 = \{\text{line, oval, rect}, \text{,}, [,]\} \cup \{0, \ldots, 9\}^+$ (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 30).
9. Upravte kód metody přímého stavového programování použitý na načítání čísel (celý kód začíná na straně 30) 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:

```
type
  TSymbol = record
    typ:     TTypSymbolu;        // identifikace symbolu
    cislo:   integer;            // atribut ve formátu celého čísla
    retezec: string;             // atribut ve formátu řetězce
  end;
```

   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

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.

Vstup:	*posloupnost 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* $y := 3$ *THEN* $y := -3$*)*

Tabulka 3.1: Vlastnosti syntaktické analýzy

3.1 Derivační strom

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

Definice 3.1 (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 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 $A \rightarrow A_1 A_2 \ldots A_k$.*
5. *Derivační strom tvoříme zleva doprava a shora dolů, proto není třeba značit orientaci hran.*

Příklad 3.1

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

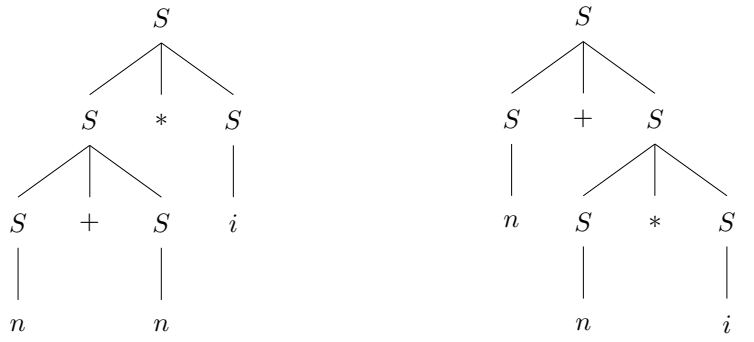
Odvodíme větu $n + n * i$ třemi různými derivacemi a ke každé vytvoříme derivační strom (v derivaci je vždy zvýrazněn ten neterminál, který má být v následujícím kroku přepsán).

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.



(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 3.1 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 (zdrojové soubory) může existovat více různých možných výstupů (derivačních stromů, syntaktických struktur programu), pak to pro programátora znamená vždy problém.

Definice 3.2 (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 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 pro zajištění jednoznačnosti překladu pomocí víceznačné gramatiky používají další možnosti, například stanovení podmínek, za jakých derivace může probíhat. Nejběžnějšími podmínkami jsou levá nebo pravá derivace.

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*.

3.2 Metody syntaktické analýzy

Při syntaktické analýze konstruujeme derivační strom pro danou větu (tj. vstup syntaktického analyzátoru, posloupnost symbolů). Podle toho, jak je konstruován derivační strom věty, rozlišujeme dvě základní metody syntaktické analýzy.

1. U metody *shora dolů* (Top-Down) derivační strom konstruujeme od kořene k listům a zleva doprava.
2. U metody *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 3.2

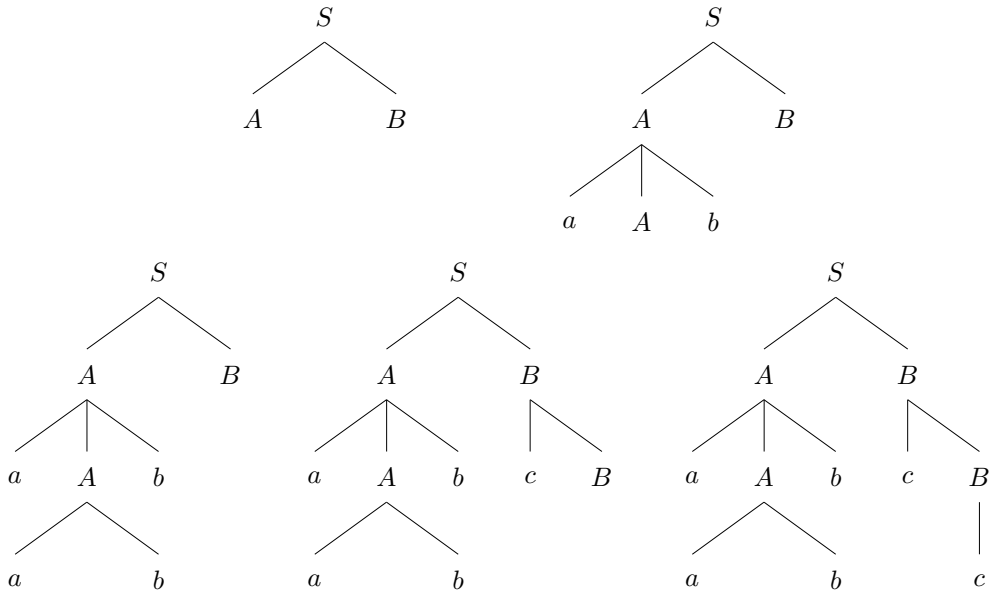
Princip obou metod si v následujících příkladech této sekce ukážeme na slově `aabbcc` odvozeném v gramatice s pravidly

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

3.2.1 Metoda shora dolů

Derivační strom věty konstruujeme shora od kořene (ohodnoceného startovacím symbolem gramatiky) dolů k listům, zleva doprava, podle levé derivace:

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



Obrázek 3.2: Postup vytvoření derivačního stromu pro levou derivaci

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 používáme vždy levou derivaci, pak tato posloupnost jednoznačně určuje celý derivační strom.

Definice 3.3 (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 3.2 bude levý rozklad slova $aabbcc$ posloupnost ①, ②, ③, ④, ⑤.

Definice 3.4 (Syntaktická analýza metodou shora dolů) *Syntaktická analýza metodou shora dolů je proces nalezení levého rozkladu dané věty.*

Účelem syntaktické analýzy věty je nalezení derivačního stromu této 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.

Pokud jen generujeme větu v gramatice, můžeme v případě více pravidel se stejnou levou stranou náhodně vybírat. Naším úkolem však bývá analýza již existující věty (programu). Zde již náhoda nepřichází v úvahu, protože posloupnost pravidel pro levou de-

rivaci již má být jednoznačná. Potřebujeme automat, který tuto analýzu provádí, a tento automat bude naprogramován tak, aby vybíral mezi pravidly vždy to správné.

Problém může nastat v případě, že máme v množině pravidel:
$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
Tato pravidla mohou být navzájem různá, nebo třeba mohou začínat stejným podřetězcem. Například podle příkladu 3.2 bychom takto váhali při zpracování větné formy `AB`. Naším úkolem (úkolem překladače) je z těchto pravidel vybrat jediné správné. Pro to existují dva základní postupy:

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 je sice účinná, ale pomalá, proto se moc 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.

3.2.2 Metoda zdola nahoru

Při použití metody zdola nahoru konstruujeme derivační strom zdola od listů nahoru ke kořeni, také postupujeme zleva doprava, protože tímto směrem se obvykle čte text nebo třeba soubor. Podle příkladu 3.2 použijeme pravou derivaci:
$S \Rightarrow AB \Rightarrow AcB \Rightarrow Acc \Rightarrow aAbcc \Rightarrow aabbcc$

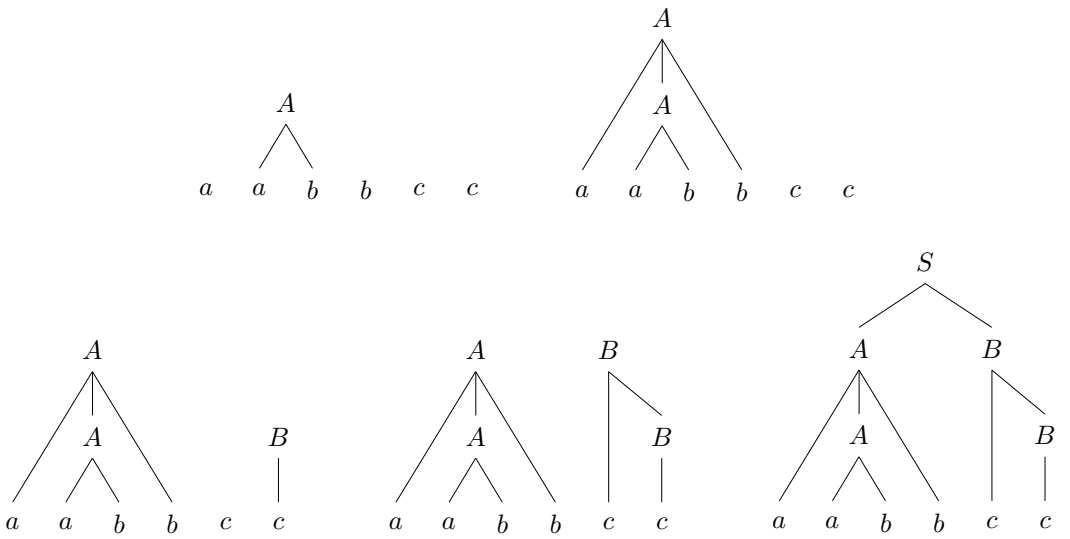
Stejně jako u první metody, i zde budeme používat lineární rozklad, tentokrát pro pravou derivaci:

Definice 3.5 (Pravý rozklad) *Pravý rozklad věty v gramatice G je **obrácená** posloupnost čísel pravidel použitých v pravé derivaci této věty v gramatice G.*

Podle příkladu 3.2 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 na straně 44.

Proč obrácená posloupnost? Při pravé derivaci v gramatice generujeme větu zprava doleva (přepisujeme vždy neterminál nejvíce vpravo), ale automat čte vstup zleva doprava,

[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 opět symbol `'a'`, tedy pro přepis `A` použijeme pravidlo ② a nikoliv ③.



Obrázek 3.3: Postup vytvoření derivačního stromu pro pravou derivaci

tedy tento postup obrací. Proto vytváří obrácenou posloupnost pravidel k té, kterou bychom použili při generování věty.

Definice 3.6 (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 musíme rozhodovat mezi pravidly, která chceme použít. Tentokrát však nejde o pravidla se stejnou levou stranou (pro stejný neterminál), ale rozhodujeme se mezi pravidly, která mají podobnou pravou stranu a jsou proto použitelná pro tentýž podřetězec větné formy. 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 na pravé straně pravidla a pokračujeme v konstrukci derivačního stromu. Když zjistíme, že tento krok nevede k úspěchu, vyzkoušíme jiný podřetězec, . . . Tato metoda je jako v předchozím případě také časově náročná, proto není moc používaná.
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.

3.3 Pomocné množiny pro syntaktickou analýzu

3.3.1 Množiny FIRST a FOLLOW

Pro analýzu vlastností gramatiky jazyka a také pro konstrukci automatu budeme používat množiny FIRST („první“) a FOLLOW („následník, následující“). Jsou to množiny terminálních symbolů definované takto:

Definice 3.7 (Množiny FIRST) *Označme α libovolnou větnou formu generovanou gramatikou G. Potom* $\mathrm{FIRST}(\alpha)$ *je množina terminálních symbolů, jimiž začínají řetězce derivované z α. Pokud existuje derivace $\alpha \Rightarrow^* \varepsilon$, pak $\varepsilon \in$* $\mathrm{FIRST}(\alpha)$.

Definice 3.8 (Množiny FOLLOW) *Nechť A je libovolný neterminál gramatiky G. Potom do množiny* $\mathrm{FOLLOW}(A)$ *řadíme právě ty terminální symboly a gramatiky G, 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* $\mathrm{FOLLOW}(A)$ *řadíme také symbol ukončení vstupního řetězce (reprezentující například konec zdrojového souboru), který budeme značit* \$.

Do množiny FIRST řadíme všechny terminály, kterými může začínat některý řetězec derivovaný z testované větné formy, do množiny FOLLOW všechny terminály, které v různých derivacích mohou následovat za testovaným neterminálem.

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.

Nejdřív se podíváme na algoritmus pro výpočet množiny FIRST. 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 $G = (N, T, P, S)$:

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

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

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

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

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 3.3

Je dána gramatika $G = (N, T, P, S)$ s těmito pravidly:
$S \rightarrow aAb \mid BAcB \mid \varepsilon$
$A \rightarrow faBd \mid aS \mid \varepsilon$
$B \rightarrow bcB \mid d$

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

Algoritmus pro výpočet množin FOLLOW je 3-krokový a počítáme vždy tyto množiny pro všechny neterminály gramatiky *najednou*.

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

$$\mathrm{FIRST}(\beta) \subseteq \mathrm{FOLLOW}(B), \quad \text{kde } A \rightarrow \alpha B\beta \tag{3.6}$$

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

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

Postup popíšeme také slovy:

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

2) Pro každé pravidlo ve tvaru $A \rightarrow \alpha B\beta$ umístíme všechny prvky množiny $\mathrm{FIRST}(\beta)$ kromě ε do $\mathrm{FOLLOW}(B)$.

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

3) Pro každé pravidlo $A \rightarrow \alpha B$ nebo $A \rightarrow \alpha B\beta$, kde existuje odvození $\beta \Rightarrow^* \varepsilon$, do množiny $\mathrm{FOLLOW}(B)$ zařadíme všechny prvky $\mathrm{FOLLOW}(A)$.

Tento krok provádíme opět tak, že procházíme všechna pravidla, oproti předchozímu bodu však hledáme neterminály, které jsou posledními symboly pravidel nebo se na konec řetězce pravidla mohou dostat v nějaké derivaci (symbol patří do množiny N_1 definované v postupu výpočtu množin FIRST).

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 samozřejmě 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 3.4

Generování množin FOLLOW ukážeme na gramatice z příkladu 3.3 s těmito pravidly:

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

Podrobný postup:

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

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

Pro přehlednost uvádíme celé množiny:

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

3.3.2 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 $k = 1$.

Definice 3.9 (Množiny $\mathbf{FIRST_k}$) *Nechť α je větná forma gramatiky G.* $\mathrm{FIRST}_k(\alpha)$ *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* $\mathrm{FIRST}_k(\alpha)$.

Definice 3.10 (Množiny $\mathbf{FOLLOW_k}$) *Nechť A je libovolný neterminál gramatiky G. Potom* $\mathrm{FOLLOW}_k(A)$ *je množina všech terminálních řetězců α gramatiky G, které se mohou vyskytovat bezprostředně vpravo od A v nějaké větné formě, tedy existuje derivace $S \Rightarrow^* \beta A\alpha\gamma$, $\alpha \in T^k$. Pokud lze v G odvodit větnou formu, ve které je délka terminálního řetězce za A menší než k, potom také tento řetězec řadíme do* $\mathrm{FOLLOW}_k(A)$. *Takový kratší řetězec ukončíme symbolem konce vstupu* \$.

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

Symbolicky můžeme konstrukci množin FIRST_k zapsat takto:

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

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

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

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

Řádek (3.11) postupu znamená rekurzivní „přenos“ algoritmu na řetězce, na které je neterminál přepisován, první tři řádky slouží k zastavení rekurze.

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

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

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

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

   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^A vytvořený v předchozím kroku nahradíme množinou řetězců $\mathrm{FIRST}_k(\beta \cdot \mathrm{FOLLOW}_k(A))$, 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 \$.

Příklad 3.5

Gramatika G je definována těmito pravidly:

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

$\mathrm{FIRST}(\varepsilon) = \{\varepsilon\}$
$\mathrm{FIRST}(ab) = \{a\}$
$\mathrm{FIRST}(cB) = \{c\}$
$\mathrm{FIRST}(Ba) = \{c, a\}$
$\mathrm{FIRST}(ABa) = \{a, c\}$

$\mathrm{FOLLOW}(S) = \{\$\}$
$\mathrm{FOLLOW}(A) = \{c, a\}$
$\mathrm{FOLLOW}(B) = \{a\}$

$\mathrm{FIRST}_2(\varepsilon) = \{\varepsilon\}$
$\mathrm{FIRST}_2(ab) = \{ab\}$
$\mathrm{FIRST}_2(cB) = \{cc, c\}$
$\mathrm{FIRST}_2(Ba) = \{cc, ca, a\}$
$\mathrm{FIRST}_2(ABa) = \{ab, cc, ca, a\}$

$\mathrm{FOLLOW}_2(S) = \{\$\}$
$\mathrm{FOLLOW}_2(A) = \{cc, ca, a^S\} = \{cc, ca, a\$\}$
$\mathrm{FOLLOW}_2(B) = \{a^S\} = \{a\$\}$

$\mathrm{FIRST}_3(ab) = \{ab\}$
$\mathrm{FIRST}_3(cB) = \{ccc, cc, c\}$
$\mathrm{FIRST}_3(Ba) = \{ccc, cca, ca, a\}$
$\mathrm{FIRST}_3(ABa) = \{abc, aba, ccc, cca, ca, a\}$

$\mathrm{FOLLOW}_3(S) = \{\$\}$
$\mathrm{FOLLOW}_3(A) = \{ccc, cca, ca^S, a^S\} = \{ccc, cca, ca\$, a\$\}$
$\mathrm{FOLLOW}_3(B) = \{a^S\} = \{a\$\}$

Příklad 3.6

Použijeme stejnou gramatiku, na které jsme v příkladech 3.3 a 3.4 ukázali postup vytváření množin FIRST a FOLLOW.

Gramatika má tato pravidla:

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

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

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

$\text{FOLLOW}_2(S) = \{\$, b^S, cb, cd\} = \{\$, b\$, bb, bc, cb, cd\}$
$\text{FOLLOW}_2(A) = \{b^S, cb, cd\} = \{b\$, bb, bc, cb, cd\}$
$\text{FOLLOW}_2(B) = \{fa, aa, af, ac, cb, cd, d^A, \$, b^S\} =$
$= \{fa, aa, af, ac, cb, cd, db, dc, \$, b\$, bb, bc\}$

U množin FOLLOW_2 byl řetězec b^S rozvinut na množinu řetězců $\{b\$, bb, bc\}$ (nesmíme zapomenout na zřetězení se sebou samým, proto zde bude i řetězec bb), řetězec d^A je rozvinut na množinu $\{db, dc\}$.

Čím vyšší je číslo k, tím více je informace, kterou množiny obsahují: podle příkladu 3.5 množina $\text{FIRST}(Ba)$ dokáže zjistit, že z řetězce Ba lze derivovat terminální řetězce začínající symboly c a a. Množina $\text{FIRST}_2(Ba)$ omezí řetězce začínající c pouze na ty, jejichž druhý symbol je c nebo a (tedy začínají cc nebo ca) a vyloučí například řetězce začínající cb, což množina pro $k = 1$ ještě nedokázala.

3.4 $LL(k)$ překlady

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.

3.4.1 $LL(k)$ gramatiky

Definice 3.11 ($LL(k)$ gramatika) *Gramatika 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 3.12 ($LL(k)$ 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*

$S \Rightarrow^* wA\alpha \Rightarrow w\beta\alpha \Rightarrow^* wx$
$S \Rightarrow^* wA\alpha \Rightarrow w\gamma\alpha \Rightarrow^* wy$

takových, že $\mathrm{FIRST}_k(x) = \mathrm{FIRST}_k(y)$, vždy platí $\beta = \gamma$.

Poznámka: Obě množiny FIRST_k terminálních řetězců x a y jsou jednoprvkové, 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 3.1 *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*

$$\mathrm{FIRST}_k(\beta \cdot \alpha) \cap \mathrm{FIRST}_k(\gamma \cdot \alpha) = \emptyset. \tag{3.12}$$

Důkaz: Podle definice 3.12 platí implikace $(\mathrm{FIRST}_k(x) = \mathrm{FIRST}_k(y)) \Longrightarrow (\beta = \gamma)$. Tuto implikaci upravíme podle pravidla $(A \to B) \iff (\neg B \to \neg A)$ do následujícího tvaru a provedeme další ekvivalentní úpravy:

$$\begin{aligned} \neg(\beta = \gamma) &\implies \neg(\mathrm{FIRST}_k(x) = \mathrm{FIRST}_k(y)) \\ \beta \neq \gamma &\implies \mathrm{FIRST}_k(x) \neq \mathrm{FIRST}_k(y) \end{aligned} \tag{3.13}$$

Vztah (3.13) 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 (3.13) platí i pro řetězce $\beta \cdot \alpha$ a $\gamma \cdot \alpha$, čímž dostáváme vztah

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

Věta 3.1 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.

3.4.2 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 3.13 (Silná LL(k) gramatika) *Nechť G 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*

$$\text{FIRST}_k(\alpha \cdot \text{FOLLOW}_k(A)) \ \cap \ \text{FIRST}_k(\beta \cdot \text{FOLLOW}_k(A)) \ = \ \emptyset \qquad (3.14)$$

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

Příklad 3.7

Zjistěte, zda je tato gramatika silná $LL(k)$ pro nějaké číslo k.

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

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

Gramatika není silná $LL(1)$.

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

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

$\text{FIRST}_2(aAc \cdot \text{FOLLOW}_2(B)) \cap \text{FIRST}_2(d \cdot \text{FOLLOW}_2(B)) = \{aa, ac\} \cap \{da, d\$\} = \emptyset$
Gramatika je silná $LL(2)$.

Příklad 3.8

Následující gramatika generuje stejný jazyk jako ta, se kterou jsme pracovali v příkladu 3.7, ale má jiná pravidla (vlastně vznikla z předchozí gramatiky ekvivalentními úpravami):

$S \to aB \mid daC \mid \varepsilon$	$\text{FOLLOW}(S) = \{\$\}$
$A \to aA \mid \varepsilon$	$\text{FOLLOW}(A) = \{c, \$\}$
$B \to bAc \mid AcaC$	$\text{FOLLOW}(B) = \{\$\}$
$C \to aAcA \mid dA$	$\text{FOLLOW}(C) = \{\$\}$

$\text{FIRST}(aB \cdot \text{FOLLOW}(S)) \cap \text{FIRST}(daC \cdot \text{FOLLOW}(S)) = \emptyset$
$\text{FIRST}(aB \cdot \text{FOLLOW}(S)) \cap \text{FIRST}(\varepsilon \cdot \text{FOLLOW}(S)) = \emptyset$
$\text{FIRST}(daC \cdot \text{FOLLOW}(S)) \cap \text{FIRST}(\varepsilon \cdot \text{FOLLOW}(S)) = \emptyset$
$\text{FIRST}(aA \cdot \text{FOLLOW}(A)) \cap \text{FIRST}(\varepsilon \cdot \text{FOLLOW}(A)) = \emptyset$
$\text{FIRST}(bAc \cdot \text{FOLLOW}(B)) \cap \text{FIRST}(AcaC \cdot \text{FOLLOW}(B)) = \{b\} \cap \{a, c\} = \emptyset$
$\text{FIRST}(aAcA \cdot \text{FOLLOW}(C)) \cap \text{FIRST}(dA \cdot \text{FOLLOW}(C)) = \emptyset$
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 3.7 a 3.8 je proto silný $LL(1)$.

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.

3.5 $LL(1)$ překlady

3.5.1 $LL(1)$ gramatika

Definice 3.14 ($LL(1)$ gramatika) *Gramatika je typu* $LL(1)$*, jestliže každá množina pravidel se stejnou levou stranou* $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ *má tyto vlastnosti:*

- vlastnost FF (FIRST, FIRST):
  pro všechna $i \neq j$ *platí* $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$,
- vlastnost FFL (FIRST, FOLLOW):
  Je-li pro nějaké i $\alpha_i \Rightarrow^* \varepsilon$*, platí pro všechna* $j \neq i$ $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$.

Pravidla testujeme opět po dvojicích. První vlastnost znamená, že pravidla se stejnou levou stranou jsou odlišitelná již prvním symbolem řetězce terminálních symbolů, který můžeme dostat derivací z řetězce těchto pravidel, druhá vlastnost znamená totéž s ohledem na ε-pravidla.

Druhou vlastnost testujeme jen tehdy, jestliže pro daný přepisovaný neterminál existuje ε-pravidlo.

Zjištění těchto vlastností si ukážeme na gramatice z příkladu 3.8.

Příklad 3.9

$S \to aB \mid daC \mid \varepsilon$ $\quad$ $\mathrm{FOLLOW}(S) = \{\$\}$
$A \to aA \mid \varepsilon$ $\quad$ $\mathrm{FOLLOW}(A) = \{c, \$\}$
$B \to bAc \mid AcaC$ $\quad$ $\mathrm{FOLLOW}(B) = \{\$\}$
$C \to aAcA \mid dA$ $\quad$ $\mathrm{FOLLOW}(C) = \{\$\}$

Neterminál S:
FF: $\mathrm{FIRST}(aB) \cap \mathrm{FIRST}(daC) = \emptyset$
$\mathrm{FIRST}(aB) \cap \mathrm{FIRST}(\varepsilon) = \emptyset$
$\mathrm{FIRST}(daC) \cap \mathrm{FIRST}(\varepsilon) = \emptyset$
FFL: $\mathrm{FIRST}(aB) \cap \mathrm{FOLLOW}(S) = \emptyset$
$\mathrm{FIRST}(daC) \cap \mathrm{FOLLOW}(S) = \emptyset$

Neterminál A:
FF: $\mathrm{FIRST}(aA) \cap \mathrm{FIRST}(\varepsilon) = \emptyset$
FFL: $\mathrm{FIRST}(aA) \cap \mathrm{FOLLOW}(A) = \emptyset$

Neterminál B:
FF: $\mathrm{FIRST}(bAc) \cap \mathrm{FIRST}(AcaC) = \emptyset$
FFL: Netestujeme, není zde ε-pravidlo.

Neterminál C:
FF: $\mathrm{FIRST}(aAcA) \cap \mathrm{FIRST}(dA) = \emptyset$
FFL: Netestujeme, není zde ε-pravidlo.

Věta 3.2 *Každá $LL(1)$ gramatika je silná $LL(1)$ gramatika.*

Důkaz: Vyplývá přímo z definice silné gramatiky (všechny řetězce v množině FIRST jsou jednoznakové) a z definice (obecné) $LL(k)$ gramatiky (v kapitole 3.4.1 na straně 51). □

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

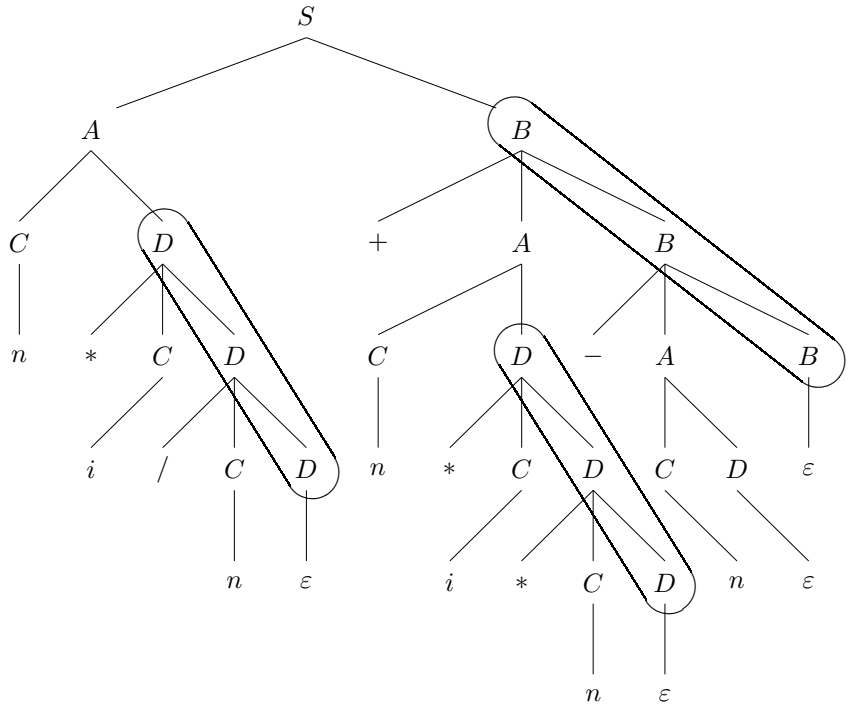
$$\mathrm{FIRST}(\alpha \cdot \mathrm{FOLLOW}(A)) \ \cap \ \mathrm{FIRST}(\beta \cdot \mathrm{FOLLOW}(A)) \ = \ \emptyset.$$

Příklad 3.10

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.

$S \to AB$	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

Pokud vytvoříme derivační strom pro větu $n * i/n + n * i * n - n$, 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 3.4 včetně vyznačení větví, ve kterých se generují skupiny podvýrazů s operátory se stejnou prioritou.



Obrázek 3.4: Derivační strom matematického výrazu v $LL(1)$ gramatice

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.

3.5.2 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.

Základní metody pro transformaci jsou následující:

Odstraníme levou rekurzi. 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:
$A \to \beta_1 B \mid \beta_2 B \mid \ldots \mid \beta_m B$
$B \to \alpha_1 B \mid \alpha_2 B \mid \ldots \mid \alpha_n B \mid \varepsilon$

Postup se může zdát nepravděpodobný. Stačí si však uvědomit, že 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.

Faktorizujeme pravidla. 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 a zavedeme nový neterminál:
$A \to \alpha B$
$B \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Eliminujeme pravidla. Někdy pomůže dosadit do pravidel za některé neterminály jejich pravé strany (můžeme tak upravit i množiny FOLLOW).

Redukujeme množiny FOLLOW. Je-li pro některý neterminál porušena vlastnost FFL, může pomoci přidání nového neterminálu, jehož množina FOLLOW převezme část prvků množiny FOLLOW konfliktního neterminálu.

Příklad 3.11

Následující gramatiku s levou rekurzí transformujeme na $LL(1)$ gramatiku.
$P \to P + Q \mid P - Q \mid Q$
$Q \to Q * R \mid Q/R \mid R$
$R \to (P) \mid i \mid n$

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.

Získali jsme tuto gramatiku:

$P \to QU$
$U \to +QU \mid -QU \mid \varepsilon$
$Q \to RV$
$V \to *RV \mid /RV \mid \varepsilon$
$R \to (P) \mid i \mid n$

Jak vidíme, vytvořená gramatika je ekvivalentní (až na označení neterminálů) s $LL(1)$ gramatikou z příkladu 3.10 na straně 55.

Příklad 3.12

Následující gramatiku transformujeme na $LL(1)$.

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

Je třeba provést faktorizaci pravidel pro neterminály S a B. Pravidla pro S upravíme takto:

$S \to AaC$
$C \to BB \mid S$

Úprava pravidel pro B je podobná. Získali jsme tuto gramatiku:

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

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.

Příklad 3.13

Následující gramatiku transformujeme na $LL(1)$.

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

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 $\mathrm{FOLLOW}(B)$ 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:

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

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

3.5.3 Překladový automat

Účelem překladu ovšem není jen popis jazyka, tedy gramatika. Potřebujeme postup, jakým z již existujícího vstupu dostaneme příslušný výstup (v našem případě zatím posloupnost čísel použitých pravidel).

Nejdřív na příkladu připomeneme postup, 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 3.10.

Příklad 3.14

Vytvoříme zásobníkový automat rozpoznávající slova jazyka generovaného gramatikou $G = (\{S, A, B, C, D\}, \{+, -, *, /, (,), n, i\}, P, S)$, množina pravidel P je v příkladu 3.10 na stránce 55.

Automat má tvar $\mathcal{A} = (\{q\}, T, T \cup N, \delta, q, S, \emptyset)$, 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 $\delta(q, a, a) = \{(q, \varepsilon)\}$:
   $\delta(q, +, +) = \{(q, \varepsilon)\}$ $\quad$ $\delta(q, (, () = \{(q, \varepsilon)\}$
   $\delta(q, -, -) = \{(q, \varepsilon)\}$ $\quad$ $\delta(q,),)) = \{(q, \varepsilon)\}$
   $\delta(q, *, *) = \{(q, \varepsilon)\}$ $\quad$ $\delta(q, n, n) = \{(q, \varepsilon)\}$
   $\delta(q, /, /) = \{(q, \varepsilon)\}$ $\quad$ $\delta(q, i, i) = \{(q, \varepsilon)\}$

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.

Automat vytvořený podle postupu z příkladu 3.14 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 v případě, že pro tentýž symbol na vrcholu zásobníku existuje více možných akcí, a samozřejmě přidáme výstupní pásku.

Definice 3.15 (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 α je nepřečtená část vstupní pásky, β 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.*

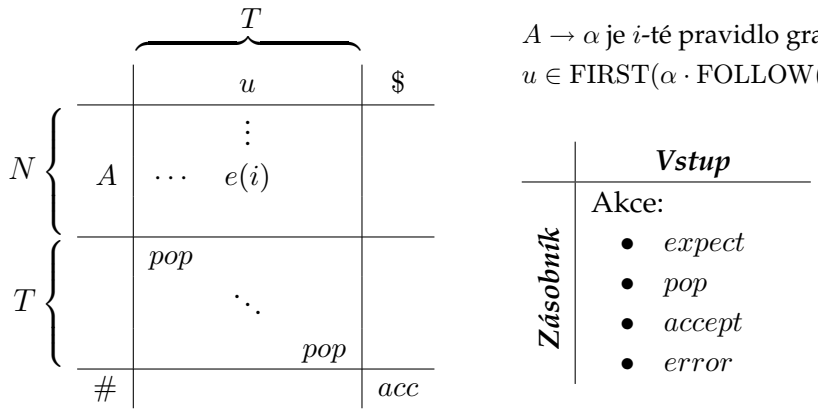
Rozkladová tabulka *automatu pro LL(1) gramatiku je zobrazení*
$M : (T \cup N \cup \{\#\}) \times (T \cup \{\$\}) \mapsto \{expand(1), \ldots, expand(n), pop, accept, error\}$,
kde jednotlivé funkční hodnoty mají tento význam:

- $expand(i)$, $1 \leq i \leq n$: *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 $error$.

V tabulce 3.2 je postup zobrazen na schématech. Názvy funkcí $expand$ a $accept$ jsou zkráceny na e a acc, v prázdných buňkách tabulky je funkce $error$.



Tabulka 3.2: Schémata rozkladové tabulky pro $LL(1)$ gramatiku

Vytvoření rozkladové tabulky a práci s ní ukážeme na gramatice z příkladů 3.8 a 3.9, 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 3.15

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 opravdu typu $LL(1)$, tak i při konstrukci rozkladové tabulky.

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

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

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 (nepřečtená část vstupní pásky je prázdná, je 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 3.16

Sestrojíme rozkladovou tabulku ke gramatice generující matematické výrazy z příkladu 3.10 na straně 55.

$S \to AB$	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.

3.5.4 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. Analýza probíhá takto:

1. Zavoláme funkci `Lex()` (to provedeme už před jakoukoliv další akcí, jeden symbol bude tedy „přednačten“). Informace o symbolu uložíme do globální proměnné datového typu `TSymbol`, a to typ symbolu i atributy.

   Jiným možným řešením je samozřejmě typ a atribut symbolu předávat v parametrech této funkce.

2. Další analýza vstupu probíhá podle pravidel gramatiky s využitím zásobníku, přesně podle rozkladové tabulky. Účelem syntaktické analýzy je vygenerovat podle gramatiky takový terminální řetězec, který by souhlasil s řetězcem na vstupu, postup generování derivace (v našem případě levý rozklad reprezentující derivační strom) je zachycen na výstupní pásce.

3. 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),

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 (procedury) odpovídající příslušným akcím z rozkladové tabulky:

- `expand(číslo_pravidla)` provede expanzi pravidla s daným číslem v zásobníku (uloží pravou stranu pravidla do zásobníku) a na výstup přidá číslo pravidla, levá strana pravidla již byla vyjmuta v jiné funkci,
- `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` (prázdná buňka),
  - 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).

Postup implementace si ukážeme na následujícím příkladu.

Příklad 3.17

Naprogramujeme syntaktickou analýzu podle gramatiky z příkladu 3.16 na straně 62 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

```
type
  TTypSymbolu = (S_ID, S_NUM, S_PLUS, S_MINUS,   // terminální symboly
    S_MUL, S_DIV, S_LPAR, S_RPAR, S_ENDOFFILE,
    S_NS, S_NA, S_NB, S_NC, S_ND, S_HASH);       // neterminální symboly

  TSymbol = record
    typ: TTypSymbolu;          // identifikace (název) symbolu
    atrib: string;             // atribut
  end;

  TZnak = record
    rad:    string;     // zpracovávaný řádek
    pozice: byte;       // pozice posledního načteného znaku na řádku
    delka:  byte;       // délka tohoto řádku
    cislo:  word;       // číslo řádku
  end;

var
  konec:        boolean;     // indikátor ukončení výpočtu, proveden accept
  znak:         TZnak;       // aktuální znak načtený ze zdroje pro funkci Lex()
  symbol:       TSymbol;     // aktuální symbol načtený z proměnné vstup
  vrchol_zas:   TTypSymbolu; // symbol na vrcholu zásobníku
  zasobnik:     TZasobnik;   // zásobník, prvky jsou typu TTypSymbolu

procedure expand(cislo_prav: integer);
begin
  case cislo_prav of
    1: begin                                    // S → AB
      Pridej_do_zasobniku(S_NB);
      Pridej_do_zasobniku(S_NA);
    end;
    2: begin                                    // A → CD
      Pridej_do_zasobniku(S_ND);
      Pridej_do_zasobniku(S_NC);
    end;
    3: begin                                    // B → +AB
      Pridej_do_zasobniku(S_NB);
      Pridej_do_zasobniku(S_NA);
```

```
      Pridej_do_zasobniku(S_PLUS);
    end;
    4: begin                                    // B → -AB
      Pridej_do_zasobniku(S_NB);
      Pridej_do_zasobniku(S_NA);
      Pridej_do_zasobniku(S_MINUS);
    end;
    ... // pro každé pravidlo gramatiky kromě epsilonových pravidel
  end;
  vystup(cislo_prav);             // zápis čísla použitého pravidla na výstup
end;

procedure error(const hlaska: string);
// Chyba syntaxe; vypíše číslo řádku, pozici na řádku a řetězec s daným hlášením
begin
  Konec := true;
  writeln('Chyba při syntaktické analýze na řádku ',znak.cislo,
    ', sloupci ',znak.pozice,': ',hlaska);
end;

procedure pop;
begin
  if symbol.typ = vrchol_zas)
    then Lex                 // lexikální analyzátor načte další symbol
    else error('chybný symbol na vstupu - '+VypisTyp(symbol.typ));
end;

procedure accept;
begin
  Konec := true;
end;
```

Dále potřebujeme inicializační a úklidovou funkci (proceduru), a také „řídicí“ proceduru, kterou jsme pojmenovali `Akce`. Celá syntaktická analýza (a s ní simultánně i lexikální) proběhne po volání hlavní funkce syntaktické analýzy pojmenované `S_analyza`.

```
procedure Init;
begin
  ...                             // inicializace vstupu a výstupu
  Vytvor_zasobnik;
  Pridej_do_zasobniku(S_HASH);    // symbol konce zásobníku
  Pridej_do_zasobniku(S_NS);      // startovací symbol gramatiky
  Lex;                            // načte symbol ze vstupu do sym
  Konec := false;
end;

procedure Done;
begin
  Zlikviduj_zasobnik;             // uvolní paměť zabranou zásobníkem
  ...                             // uzavření vstupu a výstupu
end;
```

```
procedure Akce;
begin
  vrchol_zas := Vyjmi_ze_zasobniku;
  case vrchol_zas of
    S_NS: if (symbol.typ in [S_ID,S_NUM,S_LPAR] then expand(1)
      else error('chybný symbol na vstupu - '+symbol.typ);
    S_NA: if (symbol.typ in [S_ID,S_NUM,S_LPAR] then expand(2)
      else error('chybný symbol na vstupu - '+symbol.typ);
    S_NB: case symbol.typ of
      S_PLUS: expand(3);
      S_MINUS: expand(4);
      S_RPAR,S_ENDOFFILE: expand(5);
      else error('chybný symbol na vstupu - '+symbol.typ);
    end;
    S_NC: case symbol.typ of
      S_ID: expand(7);
      S_NUM: expand(8);
      S_LPAR: expand(6);
      else error('chybný symbol na vstupu - '+symbol.typ);
    end;
    S_ND: case symbol.typ of
      S_MUL: expand(9);
      S_DIV: expand(10);
      S_PLUS,S_MINUS,S_RPAR,S_ENDOFFILE: expand(11);
      else error('chybný symbol na vstupu - '+symbol.typ);
    end;
    S_HASH: if (symbol.typ = S_ENDOFFILE) then accept
      else error('chybný symbol na vstupu - '+symbol.typ);
    S_PLUS,S_MINUS,S_MUL,S_DIV,S_LPAR,S_RPAR,S_ID,S_NUM: pop;
    else error('chybný symbol na vstupu - '+symbol.typ);
  end;
end;

procedure S_analyza;
begin
  Init;
  while (not Konec) do Akce;
  Done;
end;
```

Výhody metody:

- nepoužíváme rekurzi (neřešíme problém hloubky rekurze s prostorovou složitostí).

Nevýhody metody:

- u překladů zahrnujících např. matematické výrazy se hůře implementuje sémantika,
- potřebujeme zásobník.

Především vzhledem k první nevýhodě tuto metodu používáme většinou jen ke kontrole syntaxe zdrojového programu, ale za určitých okolností je použitelná i šíře. Pokud tuto metodu kombinujeme s jinou (například pro zpracování matematických výrazů použijeme další metodu), lze vyřešit i problém se sémantikou, jak zjistíme v následujících kapitolách.

3.5.5 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

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

kterou pak použijeme pro testování při rozhodování. Analýza probíhá takto:

1. Zavoláme funkci `Lex()`, čímž přednačteme jeden symbol, pak voláme podle potřeby syntaxe.
2. Postupujeme přesně tak, jakobychom konstruovali derivační strom „ručně“. Z každého neterminálu se v derivačním stromě musí vytvořit jeho potomci, mezi kterými mohou být i další neterminály. Rekurze tedy bude probíhat přes volání neterminálů.
3. V programu můžeme volat jen funkce nebo procedury, proto z neterminálů vytvoříme ekvivalentně pojmenované funkce (procedury) – třeba pro neterminál V vytvoříme funkci `V()` – a při konstrukci potomků v rámci takové funkce prostě na místě neterminálů voláme funkce příslušné těmto neterminálům, terminály pouze porovnáváme se vstupem:
   (a) když jsme v uzlu ohodnoceném neterminálem A, vytvoříme poduzly podle zvoleného pravidla $A \to \alpha$, tedy ve funkci `A()` postupně vyhodnocujeme všechny symboly řetězce α (například pro neterminály voláme jejich funkce),
   (b) tentýž postup rekurzívně uplatníme na všechny poduzly (zleva doprava), které jsou ohodnoceny neterminály,
   (c) u terminálních poduzlů pouze spustíme „kontrolní porovnání“ podobně, jako bylo u předchozí metody `pop`, s načtením dalšího symbolu ze vstupu, zde rekurze končí. U této metody se příslušná funkce tradičně nazývá `expect`.
4. Rekurzívní volání probíhá zleva doprava a shora dolů, tedy i vstup je čten zleva doprava.

[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.

5. Když skončí všechny rekurzivní výpočty pro jednotlivé větve a vstup je celý přečtený (na vstupu je $, resp. `S_ENDOFFILE`), akceptujeme vstup.

Budeme potřebovat tyto funkce (procedury):

- `Init` pro inicializaci výpočtu, `Done` pro ukončení,
- `expect` ověří shodnost symbolu na vstupu se symbolem, který je parametrem této funkce, a načte další symbol ze vstupu, programujeme podobně jako `pop` z předchozí metody,
- `S`, `A`, `B`, ... – pro každý neterminál vytvoříme stejně nazvanou funkci,
- `error` ošetří chybu, která se vyskytla při překladu.

Příklad 3.18

Opět použijeme gramatiku z příkladu 3.16 na straně 62 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 množiny FS pro jednotlivá pravidla:

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

Datové typy přejmeme od předchozí metody (typy symbolů pro neterminály však nebudeme potřebovat), a také proměnnou `symbol`. Hlavní funkce syntaktické analýzy bude jednodušší než u předchozí metody:

```
procedure S_analyza;
begin
  Init;
  S;
  Done;
end;
```

Funkce `Init` a `Done` zde nebudeme vypisovat, obsah bude podobný jako u předchozí metody (samozřejmě kromě inicializace a uvolnění zásobníku), nesmíme zapomenout na „přednačtení“ jednoho symbolu ze vstupu.

To, co v metodě přepisu rozkladové tabulky prováděla funkce `pop`, v této metodě provádí funkce tradičně nazývaná `expect` (ale můžeme si ji nazvat jinak). Porovná zpracovávaný terminál se symbolem na vstupu, a když se shodují, načte další symbol ze vstupu (tj. zavolá funkci `Lex`).

```
procedure expect(terminal: TTypSymbolu);
begin
  if symbol.typ = terminal then Lex
    else error('chybný symbol na vstupu - '+VypisTyp(symbol.typ));
end;
```

Další funkce konstruujeme podle pravidel. Protože pro téměř každý neterminál existuje více různých pravidel, kterými může být přepsán, a tedy více různých možností, jak v derivačním stromě pokračovat směrem dolů, je třeba rozhodnout, které z pravidel bude použito pro konstrukci další úrovně stromu. Pro rozhodování máme k dispozici vstup, konkrétně množiny $\mathrm{FS}(A, \alpha) = \mathrm{FIRST}(\alpha \cdot \mathrm{FOLLOW}(A))$.

Pokud symbol ze vstupu nepatří do žádné z množin FS, došlo k syntaktické chybě a je volána funkce `error`. Zde používáme funkci `VypisTyp` převádějící typ symbolu na řetězec.

```
procedure S;
begin
  if symbol.typ in [S_ID,S_NUM,S_LPAR] then begin            // S → AB
    A;
    B;
  end else error('chybný symbol na vstupu - '+VypisTyp(symbol.typ));
end;

procedure A;
begin
  if symbol.typ in [S_ID,S_NUM,S_LPAR] then begin            // A → CD
    C;
    D;
 end else error('chybný symbol na vstupu - '+VypisTyp(symbol.typ));
end;

procedure B;
begin
  case symbol.typ of
    S_PLUS: begin                                            // B → +AB
      expect(S_PLUS);
      A;
      B;
    end;
    S_MINUS:  begin                                          // B → -AB
      expect(S_MINUS);
      A;
      B;
    end;
    S_RPAR, S_ENDOFFILE:  ;                                  // B → ε
    else error('chybný symbol na vstupu - '+VypisTyp(symbol.typ));
  end;
end;
```

```
procedure C;
begin
  case symbol.typ of
    S_LPAR: begin                                          // C → (S)
      expect(S_LPAR);
      S;
      expect(S_RPAR);
    end;
    S_ID: expect(S_ID);                                    // C → i
    S_NUM: expect(S_NUM);                                  // C → n
    else error('chybný symbol na vstupu - '+VypisTyp(symbol.typ));
  end;
end;

procedure D;
begin
  case symbol.typ of
    S_MUL: begin                                           // D → *CD
      expect(S_MUL);
      C;
      D;
    end;
    S_DIV: begin                                           // D → /CD
      expect(S_DIV);
      C;
      D;
    end;
    S_PLUS,S_MINUS,S_RPAR,S_ENDOFFILE:  ;                  // D → ε
    else error('chybný symbol na vstupu - '+VypisTyp(symbol.typ));
  end;
end;
```

Výhody metody:

- není nutné vytvářet rozkladovou tabulku, třebaže množiny signatur vytvořit musíme,
- nepotřebujeme vlastní zásobník, rekurze probíhá pouze přes vzájemné volání funkcí (procedur) s použitím systémového zásobníku,
- není problém s navázáním sémantické analýzy.

Nevýhody metody:

- nemožnost určit hloubku rekurze.

Metodu rekurzivního sestupu budeme u LL překladů používat především pro snadnější napojení sémantické analýzy, zvláště pro interpretační překladače.

3.6 Silné $LL(k)$ gramatiky

3.6.1 Překladový automat pro silnou $LL(k)$ gramatiku

Definice silné $LL(k)$ gramatiky je na straně 52 v kapitole 3.4.2. Již víme, že v silné $LL(k)$ gramatice všechny dvojice pravidel se stejnou levou stranou splňují vzorec

$$\mathrm{FIRST}_k(\alpha \cdot \mathrm{FOLLOW}_k(A)) \cap \mathrm{FIRST}_k(\beta \cdot \mathrm{FOLLOW}_k(A)) = \emptyset.$$

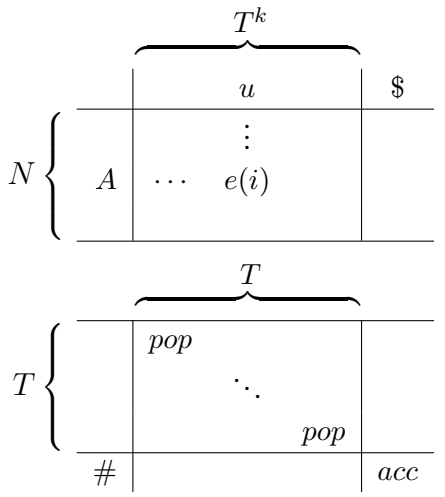
Silné $LL(k)$ gramatiky mají tyto vlastnosti:

- pro rozhodování mezi pravidly stačí nejvýše k symbolů ze vstupu,
- není třeba kontrolovat obsah zásobníku do hloubky (stačí se řídit podle jediného vyjmutého symbolu).

Proto je možné se mezi pravidly deterministicky rozhodovat podle příslušné množiny signatur délky nejvýše k – pro každé pravidlo $A \to \alpha$ vytvoříme množinu

$$\mathrm{FS}_k(A, \alpha) = \mathrm{FIRST}_k(\alpha \cdot \mathrm{FOLLOW}_k(A)).$$

Tato množina nám pomůže rozhodovat mezi pravidly se stejnou levou stranou. Na vzorci vidíme, že rozdíl oproti $LL(1)$ gramatikám je pouze v indexu k. Proto se příliš nebude lišit jak konstrukce rozkladové tabulky, tak ani implementace.



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

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

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

Tabulka 3.3 ukazuje schéma konstrukce rozkladové tabulky překladového automatu pro silnou $LL(k)$ gramatiku. Jak vidíme, je hodně podobná rozkladové tabulce pro $LL(1)$ gramatiky.

Protože při vyjímání neterminálů ze zásobníku potřebujeme pro rozhodování k-tice terminálů (navíc mohou být sloupce ohodnoceny i kratšími řetězci ukončenými symbolem \$),

musí být spodní část tabulky buď přizpůsobena, a nebo oddělena tak, jak je v tabulce naznačeno.

Příklad 3.19

Zjistíme, zda je daná gramatika $LL(1)$, když ne, tedy zda je silná $LL(k)$ pro nějaké vhodné (malé) číslo k, a sestrojíme rozkladovou tabulku.

$G = (\{S, A\},\ \{a, b\},\ P,\ S)$, množina P:

$S \to abA \mid \varepsilon$	①,②	$\mathrm{FOLLOW}(S) = \{\$, a\}$	$\mathrm{FOLLOW}_2(S) = \{\$, aa\}$
$A \to Saa \mid b$	③,④	$\mathrm{FOLLOW}(A) = \{\$, a\}$	$\mathrm{FOLLOW}_2(A) = \{\$, aa\}$

Je zřejmé, že gramatika není $LL(1)$, protože
$\mathrm{FIRST}(abA \cdot \mathrm{FOLLOW}(S)) \cap \mathrm{FIRST}(\varepsilon \cdot \mathrm{FOLLOW}(S)) = \{a\} \neq \emptyset$.

Zjistíme, zda je tato gramatika silná $LL(2)$. Otestujeme všechny dvojice pravidel se stejnou levou stranou.

$\mathrm{FIRST}_2(abA \cdot \mathrm{FOLLOW}_2(S)) \cap \mathrm{FIRST}_2(\varepsilon \cdot \mathrm{FOLLOW}_2(S)) = \{ab\} \cap \{\$, aa\} = \emptyset$
$\mathrm{FIRST}_2(Saa \cdot \mathrm{FOLLOW}_2(A)) \cap \mathrm{FIRST}_2(b \cdot \mathrm{FOLLOW}_2(A)) = \{ab, aa\} \cap \{b\$, ba\} = \emptyset$

Nyní víme, že máme vytvořit rozkladovou tabulku pro silnou $LL(2)$ gramatiku. Pro každé pravidlo zjistíme množinu $\mathrm{FS}_2(A, \alpha) = \mathrm{FIRST}_2(\alpha \cdot \mathrm{FOLLOW}_2(A))$, získanou množinu řetězců použijeme pro ohodnocení sloupců tabulky. Rozkladovou tabulku rozdělíme na dvě části, abychom se vyhnuli problémům s označením sloupců.

$\mathrm{FS}_2(S, abA) = \{ab\}$
$\mathrm{FS}_2(S, \varepsilon) = \{\$, aa\}$
$\mathrm{FS}_2(A, Saa) = \{ab, aa\}$
$\mathrm{FS}_2(A, b) = \{b\$, ba\}$

	ab	aa	$b\$$	ba	$\$$
S	e1	e2			e2
A	e3	e3	e4	e4	

	a	b	$\$$
a	*pop*		
b		*pop*	
#			*acc*

Postup zpracování slova $abaa$:

$(abaa\$,\ S\#,\ \varepsilon) \vdash (abaa\$,\ abA\#,\ 1) \vdash (baa\$,\ bA\#,\ 1) \vdash (aa\$,\ A\#,\ 1) \vdash$
$\vdash (aa\$,\ Saa\#,\ 1, 3) \vdash (aa\$,\ aa\#,\ 1, 3, 2) \vdash (a\$,\ a\#,\ 1, 3, 2) \vdash (\$,\ \#,\ 1, 3, 2)$

Postup zpracování slova aba, které nepatří do jazyka rozpoznávaného automatem:

$(aba\$,\ S\#,\ \varepsilon) \vdash (aba\$,\ abA\#,\ 1) \vdash (ba\$,\ bA\#,\ 1) \vdash (a\$,\ A\#,\ 1) \vdash error$

Příklad 3.20

Následuje trochu složitější gramatika:

$S \to aCAd \mid AjdCB \mid k$	①,②,③	$\mathrm{FOLLOW}(S) = \{\$\}$
$A \to ajAbA \mid dbBxk$	④⑤	$\mathrm{FOLLOW}(A) = \{d, j, b\}$
$B \to abCax \mid \varepsilon$	⑥⑦	$\mathrm{FOLLOW}(B) = \{x, \$\}$
$C \to dC \mid aAba \mid \varepsilon$	⑧⑨⑩	$\mathrm{FOLLOW}(C) = \{a, d, \$\}$

Je zřejmé, že to není $LL(1)$ gramatika, ke konfliktu dochází už u prvního a druhého pravidla. Zjistíme, zda je to silná $LL(2)$ gramatika.

$\text{FOLLOW}_2(S) = \{\$\}$
$\text{FOLLOW}_2(A) = \{d^S, jd, ba, bd\} = \{d\$, jd, ba, bd\}$
$\text{FOLLOW}_2(B) = \{xk, \$\}$
$\text{FOLLOW}_2(C) = \{aj, db, ab, ax, \$\}$

Vypočteme množiny signatur pro jednotlivá pravidla:

$\text{FS}_2(S, aCAd) = \{ad, aa\}$
$\text{FS}_2(S, AjdCB) = \{aj, db\}$
$\text{FS}_2(S, k) = \{k\$\}$
$\text{FS}_2(A, ajAbA) = \{aj\}$
$\text{FS}_2(A, dbBxk) = \{db\}$
$\text{FS}_2(B, abCax) = \{ab\}$
$\text{FS}_2(B, \varepsilon) = \{xk, \$\}$
$\text{FS}_2(C, dC) = \{dd, da, d\$\}$
$\text{FS}_2(C, aBba) = \{aa, ad\}$
$\text{FS}_2(C, \varepsilon) = \{aj, db, ab, ax, \$\}$

Otestování, zda je tato gramatika silná $LL(2)$, necháme na čtenáři, spočívá pouze v porovnání dvojic množin FS_2 pro stejný přepisovaný neterminál.

Vytvoříme rozkladovou tabulku. Sloupce budeme řadit podle pořadí v jednotlivých množinách FS_2. Spodní část tabulky s řádky ohodnocenými terminály a symbolem # vynecháme.

	ad	aa	aj	db	$k\$$	ab	xk	$\$$	dd	da	$d\$$	ax
S	e1	e1	e2	e2	e3							
A			e4	e5								
B						e6	e7	e7				
C	e9	e9	e10	e10		e10		e10	e8	e8	e8	e10

3.6.2 Vztah mezi silnými $LL(k)$ překlady pro různá čísla k

Rozdílnost v konstrukcích rozkladových tabulek pro různá čísla k, $k \geq 1$, ukážeme na příkladu. Uvidíme také, jak se v rozkladové tabulce projevuje chybné určení čísla k (směrem dolů).

Příklad 3.21

Podíváme se na rozdíl v rozkladových tabulkách vytvořených metodami pro $LL(1)$ a silné $LL(2)$ jazyky pro gramatiku, která není $LL(1)$.

$G = (\{S, A, B, C\}, \{a, b, c, d, f, m, p, u\}, P, S)$, kde v P jsou pravidla

$S \rightarrow fbACa \mid BaAC \mid \varepsilon$	①,②,③	$FL(S) = \{\$, a\}$
$A \rightarrow dA \mid mAp \mid \varepsilon$	④,⑤,⑥	$FL(A) = \{u, p, \$, a\}$
$B \rightarrow cBaS \mid fd$	⑦,⑧	$FL(B) = \{a\}$
$C \rightarrow uc \mid pC \mid \varepsilon$	⑨,⑩,⑪	$FL(C) = \{a, \$\}$

Je zřejmé, že tato gramatika není $LL(1)$, protože
$\text{FIRST}(fbACa \cdot \text{FOLLOW}(S)) \cap \text{FIRST}(BaAC \cdot \text{FOLLOW}(S)) = \{f\} \neq \emptyset$.

Zjistíme, zda je gramatika silná $LL(2)$. Vypočteme množiny FOLLOW_2 a FS_2 pro otestování, zda je gramatika silná $LL(2)$ a také pro konstrukci rozkladové tabulky.

$\mathrm{FOLLOW}_2(S) = \{\$, ad, am, au, ap, a^S, af, ac, a^B\} = \{\$, ad, am, au, ap, a\$, aa, af, ac\}$
$\mathrm{FOLLOW}_2(A) = \{uc, pu, pp, pa, p^A, \$, ad, am, au, ap, a^S, af, ac, a^B\} =$
$= \{uc, pu, pp, pa, p^A, \$, ad, am, au, ap, a\$, aa, af, ac\}$
$\mathrm{FOLLOW}_2(B) = \{ad, am, au, ap, a^S, af, ac, a^B\} = \{ad, am, au, ap, a\$, aa, af, ac\}$
$\mathrm{FOLLOW}_2(C) = \{a^S, \$, ad, am, au, ap, a^S, af, ac, a^B\} = \{a\$, aa, \$, ad, am, au, ap, af, ac\}$

$\mathrm{FS}_2(S, fbACa) = \{fb\}$
$\mathrm{FS}_2(S, BaAC) = \{cc, cf, fd\}$
$\mathrm{FS}_2(S, \varepsilon) = \{\$, ad, am, au, ap, a\$, aa, af, ac\}$
$\mathrm{FS}_2(A, dA) = \{dd, dm, du, dp, d\$, da\}$
$\mathrm{FS}_2(A, mAp) = \{md, mm, mp\}$
$\mathrm{FS}_2(A, \varepsilon) = \{uc, pu, pp, pa, p\$, \$, ad, am, au, ap, a\$, aa, af, ac\}$
$\mathrm{FS}_2(B, cBaS) = \{cc, df\}$
$\mathrm{FS}_2(B, fd) = \{fd\}$
$\mathrm{FS}_2(C, uc) = \{uc\}$
$\mathrm{FS}_2(C, pC) = \{pu, pp, pa, p\$\}$
$\mathrm{FS}_2(C, \varepsilon) = \{a\$, aa, \$, ad, am, au, ap, af, ac\}$

Porovnání množin FS_2 necháme na čtenáři. Rozkladová tabulka je následující:

	fb	cc	cf	fd	\$	ad	am	au	ap	$a\$$	aa	af	ac	...
S	e1	e2	e2	e2	e3	e3	e3	e3	e3	e3	e3	e3	e3	
A					e6	e6	e6	e6	e6	e6	e6	e6	e6	
B		e7	e7	e8										
C					e11	e11	e11	e11	e11	e11	e11	e11	e11	

	...	dd	dm	du	dp	$d\$$	da	md	mm	mp	uc	pu	pp	pa	$p\$$
S															
A		e4	e4	e4	e4	e4	e4	e5	e5	e5	e6	e6	e6	e6	e6
B															
C											e9	e10	e10	e10	e10

Kdybychom pro danou gramatiku chtěli sestrojit rozkladovou tabulku podle metody pro $LL(1)$ jazyky, získáme tuto tabulku:

	a	b	c	d	f	m	p	u	\$
S	e3		e2		e1, e2				e3
A	e6			e4		e5	e6	e6	e6
B			e7		e8				
C	e11						e10	e9	e11

Protože gramatika není $LL(1)$, tabulka má dvě závažné vady:

- není deterministická, pokud je v zásobníku S a na vstupu f, existují dvě různé akce,
- nejsou ošetřeny některé chyby, například pro neterminál v zásobníku S by byl podřetězec ze vstupu ab zpracován třetím pravidlem, třebaže jde o chybný vstup (na chybu by se zřejmě přišlo dále při překladu, v jiných případech by to však mohlo vést k chybnému chování překladače).

Na příkladu 3.21 vidíme, že je třeba předem důkladně otestovat gramatiku, nestačí kritérium typu „když se dá vytvořit deterministická rozkladová tabulka, tak je to v pořádku“. V tabulce $LL(1)$ sice byl nedeterminismus (pouze v jediné buňce, a to u poměrně složité gramatiky), ale může nastat případ, kdy jsou všechny buňky tabulky (zdánlivě) deterministické, ale ve skutečnosti je nedeterminismus skrytý (v buňce je operace *expand* a zároveň „neviditelná“ operace *error*).

Pro kontrolu správnosti konstrukce množin FOLLOW a FOLLOW_2 můžeme využít faktu, že v množinách FOLLOW_2 jsou pouze takové řetězce, které začínají některým z terminálů z ekvivalentní množiny FOLLOW, tedy pokud by se do množiny $\text{FOLLOW}_2(S)$ dostal řetězec začínající terminálem f, který není ve $\text{FOLLOW}(S)$, musí být někde chyba, případně když do $\text{FOLLOW}_2(A)$ nezařadíme žádný řetězec začínající terminálem p, třebaže tento terminál je ve $\text{FOLLOW}(A)$, opět budeme hledat chybu. Tentýž vztah je i mezi množinami FS a FS_2.

3.6.3 Implementace

Implementace překladu popsaného silnou $LL(k)$ gramatikou je téměř stejná jako u $LL(1)$ překladu. Rozdíl je pouze v práci se vstupem, kdy potřebujeme mít přednačtených k symbolů ze vstupu a při každém rozhodování mezi pravidly se rozhodujeme podle těchto k (někdy méně) symbolů.

Nejdřív je třeba se rozhodnout, jak vlastně bude k-tice symbolů ze vstupu reprezentována. Je více možností, například:

- pole o délce k (nebo $k+1$) prvků typu `TSymbol`, při každém volání funkce `Lex()` se v poli všechny prvky posunou o jeden doleva a nový se přidá na konec,
- kruhový seznam (statický nebo dynamický) k prvků typu `TSymbol`, v ukazateli máme zachycen momentální začátek seznamu (nebo v číselné proměnné její index), při volání funkce `Lex()` není třeba nic posouvat, pouze se přepíše původní první prvek seznamu a ukazatel se posune na následující.

Předpokládejme použití druhé možnosti.

Načrtneme implementaci oběma dříve popsanými metodami – přepisem rozkladové tabulky i rekurzivním sestupem, ošetření chyby funkcí `error` bude stejné jako v předchozích sekcích této kapitoly.

```
var
  symboly: array[1..k] of TSymbol;   // za k dosadíme skutečnou hodnotu!
  s_prvni, s_posledni: integer;
  vrchol_zas: TTypSymbolu;
  ...

procedure Init;
begin
  ...
  s_prvni := 1;
  s_posledni := 1;
  Lex;              // načte symbol na index s_posledni (tedy 1)
  if s_posledni <> S_ENDOFFILE then begin
    inc(s_posledni);
    Lex;            // načte druhý symbol na index s_posledni (tedy 2)
  end;
  // předchozí čtyři řádky celkem (k-1)-krát
  ...
end;
```

Načítání symbolů ze vstupu ukážeme zároveň pro obě použitelné metody:

Metoda přepisu rozkladové tabulky:

```
procedure pop;
begin
  if symboly[s_prvni] = vrchol_zas
  then begin
    if symboly[s_posledni] <>
      S_ENDOFFILE then begin
      inc(s_prvni);
      if s_prvni > k then
        s_prvni := 1;
      inc(s_posledni);
      if s_posledni > k then
        s_posledni := 1;
      Lex;
    end else begin
      inc(s_prvni);
      if s_prvni > k then
        s_prvni := 1;
    end;
  end else error(...);
end;
```

Metoda rekurzivního sestupu:

```
procedure expect(term: TTypSymbolu);
begin
  if symboly[s_prvni].typ = term
  then begin
    if symboly[s_posledni] <>
      S_ENDOFFILE then begin
      inc(s_prvni);
      if s_prvni > k then
        s_prvni := 1;
      inc(s_posledni);
      if s_posledni > k then
        s_posledni := 1;
      Lex;
    end else begin
      inc(s_prvni);
      if s_prvni > k then
        s_prvni := 1;
    end;
  end else error(...);
end;
```

Dále budeme předpokládat, že v rozkladové tabulce jsou na řádku označeném symbolem A prvky e(n) a e(m) tak, jak je naznačeno v tabulce vpravo. V metodě přepisu rozkladové tabulky se vstup používá v proceduře `Akce`:

	...	uv	xy	...
...		...	...	
A		e(n)	e(m)	
...				

```
procedure Akce;
begin
  vrchol_zas := Vyjmi_ze_zasobniku;
  case vrchol_zas of
    ...
    S_NA: case symboly[s_prvni].typ of
            u: if symboly[s_prvni+1].typ = v then expand(n)
               else ...;   // další řetězec začínající symbolem u nebo chyba
            x: if symboly[s_prvni+1].typ = y then expand(m)
               else ...;   // další řetězec začínající symbolem x nebo chyba
            ... // další buňka v řádku A
            else error(...);
          end; // case
    ...  // další řádky tabulky
    S_HASH: if (symboly[s_prvni].typ = S_ENDOFFILE) then accept else error(...);
    S_ID,S_NUM,S_PLUS: pop;
    else error(...);
  end; // case
end;
```

Podobně v metodě rekurzivního sestupu:

```
procedure A;
begin
  case symboly[s_prvni].typ of
    u: if symboly[s_prvni+1].typ = v then begin
         ...   // zpracování pravidla číslo n
       end else ...    // další řetězec začínající symbolem u nebo chyba
    x: if symboly[s_prvni+1].typ = y then begin
         ...   // zpracování pravidla číslo m
       end else ...    // další řetězec začínající symbolem x nebo chyba
    ...    // další buňka v řádku A
    else error(...);
  end; // case
end;
```

3.7 $LR(k)$ překlady

Zkratka $LR(k)$ znamená:

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

3.7.1 $LR(k)$ gramatiky

První uvedená definice je pouze ilustrativní, až následující definice budou mít konstrukční charakter.

Definice 3.16 ($LR(k)$ gramatika) *Gramatika je typu $LR(k)$, jestliže ji lze použít pro deterministickou syntaktickou analýzu metodou zdola nahoru (vytváříme pravý rozklad) a při rozhodování mezi pravidly potřebujeme znát nejvýše k symbolů ze vstupu.*

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

Definice 3.17 ($LR(k)$ gramatika) *Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. G je $LR(k)$ gramatika pro nějaké celé nezáporné číslo k, jestliže v případě existence dvou pravých derivací*

$S \Rightarrow^* \alpha Ax \Rightarrow \alpha\gamma x$
$S \Rightarrow^* \beta By \Rightarrow \beta\gamma y$

takových, že $\mathrm{FIRST}_k(x) = \mathrm{FIRST}_k(y)$, vždy platí $\alpha A = \beta B$.

To znamená, že pro deterministickou analýzu $LR(k)$ gramatik nám stačí v každém kroku znát kromě obsahu zásobníku nejvýše k symbolů ze vstupu.

Množiny LL a LR gramatik se částečně překrývají. Existují gramatiky, které patří do obou těchto množin, ale také gramatiky, které patří jen do jedné z nich nebo do žádné. Je zajímavé, že třída jazyků typu $LL(1)$ je vlastní podmnožinou třídy jazyků typu $LR(1)$[3].

Analýza tohoto typu gramatik se liší od předchozích LL překladů především tím, že v derivaci, používaných pravidlech i derivačním stromě postupujeme opačným směrem. Zatímco u LL gramatik jsme při vyjmutí neterminálu ze zásobníku nahrazovali tento neterminál pravou stranou některého pravidla, které tento neterminál přepisovalo, nyní budeme naopak pravou stranu pravidla (řetězec) nahrazovat levou stranou (neterminálem). Místo *expanze* budeme provádět *redukci*.

Další rozdíl je v rozhodování – nebudeme se rozhodovat mezi pravidly se stejnou levou stranou (tj. pro tentýž neterminál), ale mezi neterminály, které lze přepsat na tentýž řetězec (případně se shodují zakončení pravých stran pravidel). Stejně jako u LL překladů, i zde budeme požadovat, aby bylo možné provést deterministickou syntaktickou analýzu, tedy rozhodovat deterministicky.

Dále budeme používat rozšířenou gramatiku, především z důvodu snadnějšího odlišení kořene derivačního stromu od dalších uzlů:

Definice 3.18 (Rozšířená gramatika) *Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. Rozšířená gramatika ke gramatice G je gramatika $G' = (N', T', P', S')$, kde je $N' = N \cup \{S'\}$, $S' \notin N$, $T' = T \cup \{\#\}$, $P' = P \cup \{S' \to \#S\}$.*

[3]Pozor, zde mluvíme o jazycích, nikolv o gramatikách. Jen málokterá gramatika typu $LL(1)$ je zároveň $LR(1)$, ale lze ji transformovat na $LR(1)$.

Rozšířenou gramatiku tedy sestrojíme tak, že přidáme nový neterminál, který zároveň použijeme jako startovací symbol, a přidáme pravidlo přepisující tento neterminál na původní startovací symbol. Aby samotná analýza probíhala bez problémů, bývá výhodné přidat také symbol $\#$, jak je v definici naznačeno. V rozšířené gramatice se startovací symbol nevyskytuje v žádné větné formě kromě prvního prvku derivace.

3.7.2 Silné $LR(k)$ gramatiky

Silná $LR(k)$ gramatika je taková gramatika, pro kterou je možné vytvořit syntaktický analyzátor vykonávající syntaktickou analýzu zdola nahoru, který využívá *pouze* informace o nejbližších k symbolech v nepřečtené části vstupního řetězce a nevyžaduje další informace ze zásobníku.

Nejdřív uvedeme několik pomocných definic a pak konstrukční definici silné $LR(k)$ gramatiky použitelnou pro testování.

Definice 3.19 (Množiny BEFORE) *V gramatice $G = (N, T, P, S)$ je pro $A \in N$ množina* $\mathrm{BEFORE}(A)$ *definována takto:*

$$\begin{aligned} \mathrm{BEFORE}(A) &= \{X \in (N \cup T) \mid S \Rightarrow^* \alpha X A \beta,\ \alpha, \beta \in (N \cup T)^*\} \cup \\ &\cup \{\# \mid S \Rightarrow^* A\beta,\ \beta \in (N \cup T)^*\} \end{aligned} \tag{3.15}$$

Narozdíl od funkce FOLLOW_k, která se také počítá z neterminálů, zde nerozlišujeme verzi pro různá čísla k, vždy se jedná o množinu symbolů (řetězců o délce 1).

Do množiny $\mathrm{BEFORE}(A)$ daného neterminálu řadíme všechny symboly (terminální i neterminální), které mohou být bezprostředně před A v některé pravé derivaci; jestliže se A nachází na začátku větné formy, řadíme zde symbol konce zásobníku $\#$ (pokud jsme gramatiku předem rozšířili, tento krok není třeba).

Stejně jako u FOLLOW_k, musíme konstruovat množiny pro všechny neterminály gramatiky zároveň (jsou navzájem závislé). Narozdíl od konstrukce množin FOLLOW_k zde máme práci jednodušší v tom, že můžeme počítat s pravou rekurzí, a tedy řetězec před neterminálem ve větné formě se zpracovává až po zpracování tohoto neterminálu, nemusíme zjišťovat, jak se neterminály nalevo dále přepisují.

Postup pro (rozšířenou) gramatiku $G = (N, T, P, S)$ můžeme popsat takto:

1) $\# \in \mathrm{BEFORE}(S)$
   Do množiny startovacího symbolu zařadíme $\#$.
2) Pro každé pravidlo $B \to \alpha X A \beta,\ A, B \in N,\ X \in (N \cup T),\ \alpha, \beta \in (N \cup T)^*$:
   $X \in \mathrm{BEFORE}(A)$
   Projdeme všechna pravidla a do množin jednotlivých neterminálů zařadíme všechny symboly (terminální i neterminální), které jim přímo předcházejí.

3) Pro každé pravidlo $B \to A\beta,\ A, B \in N,\ \beta \in (N \cup T)^*$:
$\mathrm{BEFORE}(B) \subseteq \mathrm{BEFORE}(A)$
Projdeme všechna pravidla. Pokud se některý neterminál (zde A) nachází na začátku řetězce pravidla, pak celý obsah množiny BEFORE přepisovaného neterminálu (zde B) přidáme do BEFORE symbolu A (ve směru šipky pravidla). Tento krok provádíme rekurzívně tak dlouho, dokud dochází ke změnám.

Definice 3.20 (Množiny EFF_k) *V gramatice $G = (N, T, P, S)$ pro řetězec $\alpha \in (N \cup T)^*$ pro množinu $\mathrm{EFF}_k(\alpha)$ platí*

$$\begin{aligned}\mathrm{EFF}_k(\alpha) \quad = \{ \quad & w \in T^* \mid w \in \mathrm{FIRST}_k(\alpha) \textit{ a pro pravou derivaci } \alpha \Rightarrow^* \beta \Rightarrow^* wx \\ & \textit{existuje také jiný případ než } \beta = Awx \ \} \end{aligned} \tag{3.16}$$

Množina EFF_k řetězce α je vlastně krácenou verzí množiny FIRST_k téhož řetězce. Postupujeme tak, že konstruujeme množinu FIRST_k, ale zároveň hlídáme postup, jakým se řetězce do této množiny dostávají. Jestliže je daný řetězec vygenerován takovou derivací, kde musí být použito ε-pravidlo na první symbol některé větné formy, pak tento řetězec do EFF_k nezařadíme.

Pokud řetězec, který zpracováváme, se během derivace dostane do formy $A\beta$, tedy začíná neterminálem, zjistíme, jak derivace pokračuje. Pokud lze dále pokračovat pouze pravidlem $A \to \varepsilon$ a použití jiného než ε-pravidla pro A by znamenalo rekurzivní dosazení A na začátek větné formy, pak výsledný generovaný řetězec (zkrácený na délku k) vyřadíme z EFF_k.

Příklad 3.22

K dané gramatice vypočteme množiny BEFORE a některé množiny EFF.

$S' \to \#S$	$\mathrm{BEFORE}(S') = \{\#\}$
$S \to aAB \mid \varepsilon$	$\mathrm{BEFORE}(S) = \{\#, A, b\}$
$A \to ACbA \mid \varepsilon$	$\mathrm{BEFORE}(A) = \{a, b, A\}$
$B \to bBc \mid Sm$	$\mathrm{BEFORE}(B) = \{A, b\}$
$C \to ABc \mid d \mid \varepsilon$	$\mathrm{BEFORE}(C) = \{A\}$

Porovnáme množiny FIRST a EFF některých řetězců:

$\mathrm{FIRST}(aAB) = \{a\}$	$\mathrm{FIRST}(ACbA) = \{b, a, m, d\}$
$\mathrm{EFF}(aAB) = \{a\}$	$\mathrm{EFF}(ACbA) = \emptyset$
$\mathrm{FIRST}(ABc) = \{b, a, m\}$	$\mathrm{FIRST}(C) = \{b, a, m, d, \varepsilon\}$
$\mathrm{EFF}(ABc) = \emptyset$	$\mathrm{EFF}(C) = \{d\}$

U prvního řetězce je výsledek zřejmý. Protože začíná terminálem, nemůže být na začátku větné formy použito ε-pravidlo. U dalších dvou řetězců je množina EFF prázdná, protože v pravidlech neterminálu A je levá rekurze (A je vždy na začátku řetězce) končící pouze použitím pravidla $A \to \varepsilon$.

Čtvrtý řetězec má množinu EFF sice neprázdnou, přesto jsme však některé symboly vyřadili, protože pouze přepis symbolu C pravidlem $C \to d$ umožňuje vytvořit derivaci bez levé rekurze ukončené ε-pravidlem. Slovo ε jsme vyřadili, protože do $\mathrm{FIRST}(C)$ mohlo být zařazeno pouze derivací $C \Rightarrow \varepsilon$ – opět ε-pravidlo na neterminál, který je prvním symbolem větné formy.

Teď již známe vše potřebné pro pochopení definice silné $LR(k)$ gramatiky použitelné pro testování.

Definice 3.21 (Silná $LR(k)$ gramatika) *Bezkontextová gramatika $G = (N, T, P, S)$ je silná $LR(k)$, jestliže pro podle ní sestrojenou rozšířenou gramatiku $G' = (N \cup \{S'\}, T \cup \{\#\}, P', S')$, kde $P' = P \cup \{S' \to \#S\}$, platí následující dvě podmínky:*

1. *Pro každou dvojici pravidel v P' ve tvaru*

   (a) $A \to \alpha X,\ B \to \beta X,$
   (b) $A \to \alpha X,\ B \to \varepsilon,\ X \in \mathrm{BEFORE}(B),$
   (c) $A \to \varepsilon,\ B \to \varepsilon,\ \exists X \in \mathrm{BEFORE}(A) \cap \mathrm{BEFORE}(B),$

   platí $\mathrm{FOLLOW}_k(A) \cap \mathrm{FOLLOW}_k(B) = \emptyset.$
2. *Pro každou dvojici pravidel v P' ve tvaru*

   (a) $A \to \alpha X,\ B \to \beta X \gamma,$
   (b) $A \to \varepsilon,\ B \to \beta X \gamma,\ X \in \mathrm{BEFORE}(A),$
   (c) $A \to \varepsilon,\ B \to \gamma,\ \exists X \in \mathrm{BEFORE}(A) \cap \mathrm{BEFORE}(B),$

   platí $\mathrm{FOLLOW}_k(A) \cap \mathrm{EFF}_k(\gamma \cdot \mathrm{FOLLOW}_k(B)) = \emptyset.$

Příklad 3.23

Je dána bezkontextová gramatika. Nejdřív zjistíme, zda je silná $LL(1)$.

$S \to \#E$
$E \to E + T \mid E - T \mid T$
$T \to T * F \mid T/F \mid F$
$F \to n \mid i \mid (E)$

Přímo z pravidel gramatiky je zřejmé, že to není $LL(k)$ gramatika pro žádné k, protože se v nich vyskytuje levá rekurze. Zjistíme, zda je tato gramatika silná $LR(1)$. Nejdřív vypočteme množiny FOLLOW a BEFORE.

$\mathrm{FOLLOW}(S) = \{\$\}$ $\qquad$ $\mathrm{BEFORE}(S) = \{\#\}$
$\mathrm{FOLLOW}(E) = \{+, -,), \$\}$ $\qquad$ $\mathrm{BEFORE}(E) = \{(, \#\}$
$\mathrm{FOLLOW}(T) = \{*, /, +, -,), \$\}$ $\qquad$ $\mathrm{BEFORE}(T) = \{+, -, (, \#\}$
$\mathrm{FOLLOW}(F) = \{*, /, +, -,), \$\}$ $\qquad$ $\mathrm{BEFORE}(F) = \{*, /, +, -, (, \#\}$

1. (a) testujeme pravidla $E \to E + T,\ E \to E - T$:
   $\text{FOLLOW}(E) \cap \text{FOLLOW}(E) \neq \emptyset$

Už z prvního testu vidíme, že gramatika není silná $LR(1)$.

Gramatiku z příkladu 3.23 upravíme tak, aby pravidla přepisující tentýž neterminál nekončila stejně. Použijeme jednoduchou metodu *pravé faktorizace*, tedy „vytknutí“ konce pravidla. Metoda je podobná levé faktorizaci, se kterou jsme se setkali u LL gramatik na straně 56.

Příklad 3.24

Prověříme bezkontextovou gramatiku generující matematické výrazy, která má již lepší vlastnosti z hlediska zpracování než gramatika z příkladu 3.23.

$S \to \#E$	$\text{FOLLOW}(S) = \{\$\}$	$\text{BEFORE}(S) = \{\#\}$
$E \to AT$	$\text{FOLLOW}(E) = \{+, -,), \$\}$	$\text{BEFORE}(E) = \{(, \#\}$
$A \to E+ \mid E- \mid \varepsilon$	$\text{FOLLOW}(A) = \{n, i, (\}$	$\text{BEFORE}(A) = \{(, \#\}$
$T \to BF$	$\text{FOLLOW}(T) = \{*, /, +, -,), \$\}$	$\text{BEFORE}(T) = \{A\}$
$B \to T* \mid T/ \mid \varepsilon$	$\text{FOLLOW}(B) = \{n, i, (\}$	$\text{BEFORE}(B) = \{A\}$
$F \to n \mid i \mid (E)$	$\text{FOLLOW}(F) = \{*, /, +, -,), \$\}$	$\text{BEFORE}(F) = \{B\}$

Oproti předchozí gramatice vidíme změny především v množinách BEFORE. Vypočteme potřebné množiny EFF a zjistíme, zda úpravy vedly k silné $LR(k)$ gramatice.

$\text{EFF}(+ \cdot \text{FOLLOW}(A)) = \{+\}$
$\text{EFF}(- \cdot \text{FOLLOW}(A)) = \{-\}$
$\text{EFF}(\ ) \cdot \text{FOLLOW}(F)) = \{\ )\}$
$\text{EFF}(* \cdot \text{FOLLOW}(B)) = \{*\}$
$\text{EFF}(/ \cdot \text{FOLLOW}(B)) = \{/\}$
$\text{EFF}(\#E \cdot \text{FOLLOW}(S)) = \{\#\}$
$\text{EFF}(E \cdot \text{FOLLOW}(S)) = \emptyset$
$\text{EFF}(E) \cdot \text{FOLLOW}(F)) = \emptyset$
$\text{EFF}(T \cdot \text{FOLLOW}(E)) = \emptyset$
$\text{EFF}(AT \cdot \text{FOLLOW}(E)) = \emptyset$
$\text{EFF}(BF \cdot \text{FOLLOW}(T)) = \emptyset$

Proč jsou množiny EFF řetězců začínajících na A, B, E a T prázdné? Protože všechny prvky příslušných množin FIRST lze vygenerovat pouze takovou derivací, kde je v některé větné formě na první symbol řetězce použito ε-pravidlo, například:

$AT \Rightarrow \ldots \Rightarrow A\alpha \Rightarrow E + \alpha \Rightarrow AT + \alpha \Rightarrow \ldots \Rightarrow A\beta + \alpha \Rightarrow E + \beta + \alpha \Rightarrow AT + \beta + \alpha \Rightarrow \ldots$

Dokud v této derivaci nepoužijeme na symbol A pravidlo $A \to \varepsilon$, nezbavíme se ho.

Zjistíme, zda je gramatika silná $LR(1)$. Použijeme opět definici 3.21 ze strany 82.

1. (a) Není co testovat, žádná dvě pravidla nekončí stejně.
   (b) Není co testovat, na konci žádného pravidla se nevyskytují prvky množiny $\mathrm{BEFORE}(A)$ ani $\mathrm{BEFORE}(B)$.
   (c) Není co testovat, množiny $\mathrm{BEFORE}(A)$ a $\mathrm{BEFORE}(B)$ mají prázdný průnik.
2. (a) $S \to E,\ A \to E+$ $\quad \mathrm{FOLLOW}(S) \cap \mathrm{EFF}(+ \cdot \mathrm{FOLLOW}(A)) = \emptyset$
   $S \to E,\ A \to E-$ $\quad \mathrm{FOLLOW}(S) \cap \mathrm{EFF}(- \cdot \mathrm{FOLLOW}(A)) = \emptyset$
   $S \to E,\ F \to (E)$ $\quad \mathrm{FOLLOW}(S) \cap \mathrm{EFF}(\ ) \cdot \mathrm{FOLLOW}(F)) = \emptyset$
   $E \to AT,\ B \to T*$ $\quad \mathrm{FOLLOW}(E) \cap \mathrm{EFF}(* \cdot \mathrm{FOLLOW}(B)) = \emptyset$
   $E \to AT,\ B \to T/$ $\quad \mathrm{FOLLOW}(E) \cap \mathrm{EFF}(/ \cdot \mathrm{FOLLOW}(B)) = \emptyset$
   (b) $A \to \varepsilon,\ S' \to \#E$ $\quad \mathrm{FOLLOW}(A) \cap \mathrm{EFF}(E \cdot \mathrm{FOLLOW}(S)) = \emptyset$
   $A \to \varepsilon,\ F \to (E)$ $\quad \mathrm{FOLLOW}(A) \cap \mathrm{EFF}(E) \cdot \mathrm{FOLLOW}(F)) = \emptyset$
   $B \to \varepsilon,\ E \to AT$ $\quad \mathrm{FOLLOW}(B) \cap \mathrm{EFF}(T \cdot \mathrm{FOLLOW}(E)) = \emptyset$
   (c) $A \to \varepsilon,\ S \to \#E$ $\quad \mathrm{FOLLOW}(A) \cap \mathrm{EFF}(\#E \cdot \mathrm{FOLLOW}(S)) = \emptyset$
   $A \to \varepsilon,\ E \to AT$ $\quad \mathrm{FOLLOW}(A) \cap \mathrm{EFF}(AT \cdot \mathrm{FOLLOW}(E)) = \emptyset$
   $B \to \varepsilon,\ T \to BF$ $\quad \mathrm{FOLLOW}(B) \cap \mathrm{EFF}(BF \cdot \mathrm{FOLLOW}(T)) = \emptyset$

Ve všech testech vyšly prázdné množiny, tedy gramatika je silná $LR(1)$.

3.7.3 Překladový automat pro silnou $LR(k)$ gramatiku

Definice překladového automatu pro silnou $LR(k)$ gramatiku je hodně podobná obdobné definici pro silnou $LL(k)$ gramatiku, rozdíl je především v práci se zásobníkem – místo expanze směrem dolů se provádí redukce směrem nahoru po derivačním stromě.

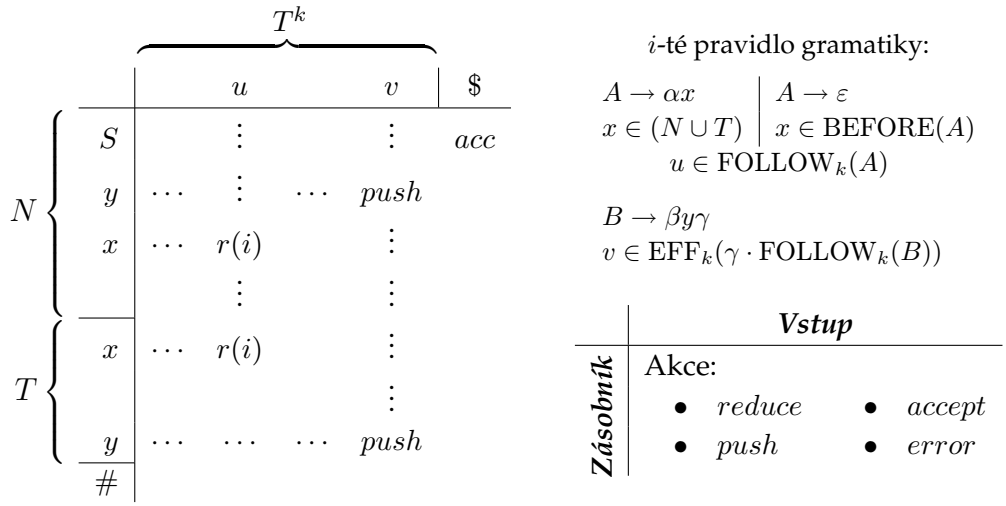
Definice 3.22 (Překladový automat pro silnou $LR(k)$ gramatiku) *Překladový automat pro silnou $LR(k)$ (rozšířenou) gramatiku $G = (N, T, P, S)$ je zásobníkový automat s jediným stavem, rozšířený o výstupní pásku a definovaný dále popsanou rozkladovou tabulkou.*

Konfigurace *překladového automatu má tvar (α, β, γ), kde α je nepřečtená část vstupní pásky, β je obsah zásobníku a γ je obsah výstupní pásky.* Počáteční konfigurace *má tvar $(w, \#, \varepsilon)$, kde w je vstupní řetězec a $\#$ symbol konce zásobníku.*

Rozkladová tabulka *automatu pro silnou $LR(k)$ gramatiku je zobrazení*
$M : (T \cup N \cup \{\#\}) \times (T \cup \{\$\})^k \mapsto \{reduce(1), \ldots, reduce(n), push, accept, error\}$,
kde jednotlivé funkční hodnoty mají tento význam:

- $reduce(i)$, $1 \leq i \leq n$: *Je-li $A \to \alpha$ i-té pravidlo gramatiky, na vrcholu zásobníku je (zdola nahoru) řetězec α, na vstupu symbol x, provede automat změnu konfigurace $(x\sigma, \alpha\phi\#, \gamma) \vdash (x\sigma, A\phi\#, \gamma i)$, tedy v zásobníku nahradí řetězec α levou stranou pravidla $A \to \alpha$, vstupní pásky si nevšímá a na výstupní pásku připíše číslo i.*

- *$push$: Automat vloží symbol ze vstupu do zásobníku, načte další symbol ze vstupu, tedy je-li na vstupu symbol x, provede změnu konfigurace $(x\sigma, \phi, \gamma) \vdash (\sigma, x\phi, \gamma)$.*
- *$accept$: K přijetí vstupu dojde, pokud je v zásobníku pouze startovací symbol gramatiky a vstup je celý přečtený. Na výstupní pásce je pravý rozklad vstupní věty.*
- *$error$: Tato hodnota znamená chybu ve výpočtu, zde jde o syntaktickou chybu.*



Tabulka 3.4: Schémata rozkladové tabulky pro silnou $LR(k)$ gramatiku

Schéma rozkladové tabulky pro silnou $LR(k)$ gramatiku je v tabulce 3.4. Postup vytvoření rozkladové tabulky pro silnou $LR(k)$ gramatiku je následující:

1. Předpokládejme, že gramatika G, kterou zpracováváme, je již rozšířená, tj. startovací symbol se nevyskytuje na pravé straně žádného pravidla.
2. Vypočteme množiny BEFORE a FOLLOW_k pro všechny neterminály gramatiky, podle potřeby množiny EFF_k.
3. Tvoříme obsah tabulky M[řádek, sloupec]:

   - vstup je akceptován bez syntaktické chyby, pokud je celý přečtený a v zásobníku je pouze startovací symbol – $M[S, \$] = accept$,
   - pro každé $A \to \alpha X$ (i-té) pravidlo a pro všechny řetězce $u \in \text{FOLLOW}_k(A)$ bude $M[X, u] = reduce(i)$ – zpracováváme poslední symboly pravých stran všech pravidel,
   - pro každé $A \to \varepsilon$ (i-té) pravidlo gramatiky, pro všechny (terminální i neterminální) symboly $X \in \text{BEFORE}(A)$ a řetězce $u \in \text{FOLLOW}_k(A)$ bude $M[X, u] = reduce(i)$ – zpracováváme ε-pravidla,

- pro každé $B \to \beta X \gamma,\ \gamma \neq \varepsilon,\ u \in \mathrm{EFF}_k(\gamma \cdot \mathrm{FOLLOW}_k(B))$ doplníme položku $M[X, u] = push$ – zpracováváme všechny symboly pravých stran, které nejsou na konci řetězce pravidla,
- jinak $M[X, u] = error$.

Operace $push$ má za úkol doplnit do zásobníku terminální symboly pravidla (neterminální symboly jsou vkládány redukcí), po doplnění celého řetězce pravidla je při zpracování jeho posledního symbolu (X) provedena redukce řetězce na neterminál v pravidle přepisovaný.

Soustředíme se na gramatiky, které jsou silné $LR(1)$. Takových gramatik je poměrně hodně, a patří zde i mnohé gramatiky popisující syntaktickou strukturu běžných programovacích jazyků.

Příklad 3.25

Zjistíme, zda je daná gramatika typu silná $LR(1)$, a pokud ano, vytvoříme rozkladovou tabulku.

$S \to aABc \mid \varepsilon$	Rozšířená:	$S' \to \#S$	⓪
$A \to Ab \mid c$		$S \to aABc \mid \varepsilon$	①,②
$B \to Bd \mid m$		$A \to Ab \mid c$	③,④
		$B \to Bd \mid m$	⑤,⑥

$\mathrm{FOLLOW}(S') = \{\$\}$	$\mathrm{BEFORE}(S') = \{\#\}$
$\mathrm{FOLLOW}(S) = \{\$\}$	$\mathrm{BEFORE}(S) = \{\#\}$
$\mathrm{FOLLOW}(A) = \{m, b\}$	$\mathrm{BEFORE}(A) = \{a\}$
$\mathrm{FOLLOW}(B) = \{c, d\}$	$\mathrm{BEFORE}(B) = \{A\}$

Ověříme, zda je gramatika silná $LR(1)$, budeme postupovat podle definice 3.21 na straně 82. Tento test je vždy potřeba provést, aby vytvořená rozkladová tabulka byla funkční.

1. (a) $S \to aABc,\ A \to c$ $\quad \mathrm{FOLLOW}(S) \cap \mathrm{FOLLOW}(A) = \emptyset$
   (b) Není co testovat.
   (c) Není co testovat.
2. (a) Není co testovat.
   (b) $S \to \varepsilon,\ S' \to \#S$ $\quad \mathrm{FOLLOW}(S) \cap \mathrm{EFF}(S \cdot \mathrm{FOLLOW}(S') = \emptyset$
   (c) $S \to \varepsilon,\ S' \to \#S$ $\quad \mathrm{FOLLOW}(S) \cap \mathrm{EFF}(\#S \cdot \mathrm{FOLLOW}(S') = \emptyset$

Gramatika je silná $LR(1)$, proto vytvoříme rozkladovou tabulku.

	a	b	c	d	m	\$
S'						acc
S						r0
A		$push$			$push$	
B			$push$	$push$		
a			$push$			
b		r3			r3	
c		r4			r4	r1
d			r5	r5		
m			r6	r6		
$\#$	$push$					r2

Podle rozkladové tabulky zpracujeme několik slov, která předem odvodíme v gramatice. Všimneme si vztahu mezi použitými pravidly v derivaci a pravým rozkladem vygenerovaným automatem, a také rozdílu oproti práci překladového automatu LL gramatik.

Gramatika:
$S' \overset{0}{\Rightarrow} S \overset{2}{\Rightarrow} \varepsilon$

Automat:
$(\$,\ \#,\ \varepsilon) \vdash (\$,\ \#S,\ 2) \vdash (\$,\ S',\ 2, 0)$

Gramatika:
$S' \overset{0}{\Rightarrow} S \overset{1}{\Rightarrow} aABc \overset{5}{\Rightarrow} aABdc \overset{6}{\Rightarrow} aAmdc \overset{4}{\Rightarrow} acmdc$

Automat:
$(acmdc\$,\ \#,\ \varepsilon) \vdash (cmdc\$,\ \#a,\ \varepsilon) \vdash (mdc\$,\ \#ac,\ \varepsilon) \vdash (mdc\$,\ \#aA,\ 4) \vdash$
$\vdash (dc\$,\ \#aAm,\ 4) \vdash (dc\$,\ \#aAB,\ 4, 6) \vdash (c\$,\ \#aABd,\ 4, 6) \vdash (c\$,\ \#aAB,\ 4, 6, 5) \vdash$
$\vdash (\$,\ \#aABc,\ 4, 6, 5) \vdash (\$,\ \#S,\ 4, 6, 5, 1) \vdash (\$,\ S',\ 4, 6, 5, 1, 0)$

Odvození slova nepatřícího do jazyka rozpoznávaného automatem vypadá takto:
$(aba\$,\ \#,\ \varepsilon) \vdash (ba\$,\ \#a,\ \varepsilon) \vdash error$

Rozdíl oproti LL překladu je už v počáteční konfiguraci – v zásobníku je pouze symbol $\#$, a v koncové konfiguraci, kde naopak musí být počáteční symbol gramatiky. Dále stav zásobníku zapisujeme naopak, aby byla pravá strana pravidla lépe vidět (symboly ve správném pořadí).

Kdyby v některém sloupci rozkladové tabulky nebyla žádná operace kromě $error$, znamenalo by to, že se tento symbol (k-tice symbolů) ve vstupu vůbec nemůže vyskytovat. Podobné pravidlo platí i o řádcích – kdyby v některém řádku nebylo možné reagovat jakoukoliv operací, znamenalo by to, že symbol, kterým je řádek označen, se nemůže vyskytovat v zásobníku, a tedy je v gramatice nadbytečný (nebo je chyba v konstrukci rozkladové tabulky).

Příklad 3.26

Vytvoříme rozkladovou tabulku pro gramatiku, o které již víme, že je silná $LR(k)$ (viz příklad 3.24 na straně 83).

$S \to \#E$ ⓪
$E \to AT$ ①
$A \to E+ \mid E- \mid \varepsilon$ ②,③,④
$T \to BF$ ⑤
$B \to T* \mid T/ \mid \varepsilon$ ⑥,⑦,⑧
$F \to n \mid i \mid (E)$ ⑨,⑩,⑪

$\text{FOLLOW}(S) = \{\$\}$
$\text{FOLLOW}(E) = \{+, -,), \$\}$
$\text{FOLLOW}(A) = \{n, i, (\}$
$\text{FOLLOW}(T) = \{*, /, +, -,), \$\}$
$\text{FOLLOW}(B) = \{n, i, (\}$
$\text{FOLLOW}(F) = \{*, /, +, -,), \$\}$

$\text{BEFORE}(S) = \{\#\}$
$\text{BEFORE}(E) = \{(, \#\}$
$\text{BEFORE}(A) = \{(, \#\}$
$\text{BEFORE}(T) = \{A\}$
$\text{BEFORE}(B) = \{A\}$
$\text{BEFORE}(F) = \{B\}$

$\text{EFF}(+ \cdot \text{FOLLOW}(A)) = \{+\}$
$\text{EFF}(- \cdot \text{FOLLOW}(A)) = \{-\}$
$\text{EFF}(\ ) \cdot \text{FOLLOW}(F)) = \{\ )\}$
$\text{EFF}(* \cdot \text{FOLLOW}(B)) = \{*\}$
$\text{EFF}(/ \cdot \text{FOLLOW}(B)) = \{/\}$

$\text{EFF}(E \cdot \text{FOLLOW}(S)) = \emptyset$
$\text{EFF}(E) \cdot \text{FOLLOW}(F)) = \emptyset$
$\text{EFF}(T \cdot \text{FOLLOW}(E)) = \emptyset$
$\text{EFF}(F \cdot \text{FOLLOW}(T)) = \{n, i, (\}$

Rozkladová tabulka:

	n	i	$+$	$-$	$*$	$/$	$($	$)$	\$
S									acc
E			$push$	$push$				$push$	r0
A	r8	r8					r8		
T			r1	r1	$push$	$push$		r1	r1
B	$push$	$push$					$push$		
F			r5	r5	r5	r5		r5	r5
n			r9	r9	r9	r9		r9	r9
i			r10	r10	r10	r10		r10	r10
$+$	r2	r2					r2		
$-$	r3	r3					r3		
$*$	r6	r6					r6		
$/$	r7	r7					r7		
$($	r4	r4					r4		
$)$			r11	r11	r11	r11		r11	r11
$\#$	r4	r4					r4		

Podle tabulky zpracujeme vstup $n + i * n$.

$(n + i * n\$,\ \#,\ \varepsilon) \vdash (n + i * n\$,\ \#A,\ 4) \vdash (n + i * n\$,\ \#AB,\ 4, 8) \vdash (+i * n\$,\ \#ABn,\ 4, 8) \vdash$
$\vdash (+i * n\$,\ \#ABF,\ 4, 8, 9) \vdash (+i * n\$,\ \#AT,\ 4, 8, 9, 5) \vdash (+i * n\$,\ \#E,\ 4, 8, 9, 5, 1) \vdash$
$\vdash (i * n\$,\ \#E+,\ 4, 8, 9, 5, 1) \vdash (i * n\$,\ \#A,\ 4, 8, 9, 5, 1, 2) \vdash (i * n\$,\ \#AB,\ 4, 8, 9, 5, 1, 2, 8) \vdash$
$\vdash (*n\$,\ \#ABi,\ 4, 8, 9, 5, 1, 2, 8) \vdash (*n\$,\ \#ABF,\ 4, 8, 9, 5, 1, 2, 8, 10) \vdash$
$\vdash (*n\$,\ \#AT,\ 4, 8, 9, 5, 1, 2, 8, 10, 5) \vdash (n\$,\ \#AT*,\ 4, 8, 9, 5, 1, 2, 8, 10, 5) \vdash$
$\vdash (n\$,\ \#AB,\ 4, 8, 9, 5, 1, 2, 8, 10, 5) \vdash (\$,\ \#ABn,\ 4, 8, 9, 5, 1, 2, 8, 10, 5) \vdash$
$\vdash (\$,\ \#E,\ 4, 8, 9, 5, 1, 2, 8, 10, 5, 1) \vdash (\$,\ S,\ 4, 8, 9, 5, 1, 2, 8, 10, 5, 1, 0) \vdash accept$

Pravý rozklad pro vstup $n + i * n$ je $4, 8, 9, 5, 1, 2, 8, 10, 5, 1$.

3.7.4 Implementace

Použijeme *metodu přepisu rozkladové tabulky*. Postup bude podobný tomu, co známe z implementace překladu $LL(1)$ gramatik, rozdíl bude především v práci se zásobníkem a se vstupem.

Analýza probíhá takto:

1. Zavoláme funkci `Lex()` („přednačteme“ jeden symbol). Typ a atribut symbolu uložíme do globální proměnné datového typu `TSymbol`, a to typ symbolu i atribut.
2. Dále postupujeme stejně, jako při práci s „papírovou“ rozkladovou tabulkou – v cyklu jsou volány funkce provádějící operace redukce pravidel v zásobníku a vkládání terminálů ze vstupu do zásobníku.
3. 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, resp. buňky obsahující chybový stav), 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. Typ chyby je vhodné co nejlépe konkretizovat.
4. Používáme zásobník, pro účely syntaktické analýzy jsou jeho prvky zatím pouze typu `TTypSymbolu`.

Potřebujeme tyto funkce (procedury) odpovídající příslušným akcím z rozkladové tabulky:

- `reduce(číslo_pravidla)` provede redukci pravidla s daným číslem v zásobníku (vyjme ze zásobníku tolik symbolů, jaká je délka pravé strany pravidla a vloží do zásobníku neterminál pravidlem přepisovaný) a na výstup přidá číslo pravidla,
- `push` vloží symbol ze vstupu do zásobníku a načte další symbol ze vstupu,

- `accept` při konci vstupu a konci zásobníku (po vyjmutí startovacího symbolu gramatiky) ukončí výpočet programu,
- `error` ošetří chybu, která se vyskytla při překladu, identifikaci chyby můžeme provést v parametru této funkce, také vypíše pozici chyby ve zdrojovém kódu,
- `Akce` v cyklu provádí tyto kroky:
  - podle momentálního vrcholu zásobníku a symbolu na vstupu určí řádek a sloupec tabulky,
  - podle obsahu buňky na daném řádku a sloupci zavolá funkci `reduce`, `push`, `accept` nebo `error` (prázdná buňka znamená `error`),
- inicializační funkce `Init` otevře všechny potřebné soubory, inicializuje zásobník (vloží symbol # – `S_HASH`) 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.

Postup implementace si ukážeme na následujícím příkladu.

Příklad 3.27

Naprogramujeme syntaktickou analýzu podle gramatiky z příkladu 3.26 na straně 88 s těmito pravidly a rozkladovou tabulkou:

$S \to \#E$	⓪
$E \to AT$	①
$A \to E+ \mid E- \mid \varepsilon$	②,③,④
$T \to BF$	⑤
$B \to T* \mid T/ \mid \varepsilon$	⑥,⑦,⑧
$F \to n \mid i \mid (E)$	⑨,⑩,⑪

$\mathrm{FOLLOW}(S) = \{\$\}$	$\mathrm{BEFORE}(S) = \{\#\}$
$\mathrm{FOLLOW}(E) = \{+, -,), \$\}$	$\mathrm{BEFORE}(E) = \{(, \#\}$
$\mathrm{FOLLOW}(A) = \{n, i, (\}$	$\mathrm{BEFORE}(A) = \{(, \#\}$
$\mathrm{FOLLOW}(T) = \{*, /, +, -,), \$\}$	$\mathrm{BEFORE}(T) = \{A\}$
$\mathrm{FOLLOW}(B) = \{n, i, (\}$	$\mathrm{BEFORE}(B) = \{A\}$
$\mathrm{FOLLOW}(F) = \{*, /, +, -,), \$\}$	$\mathrm{BEFORE}(F) = \{B\}$

	n	i	$+$	$-$	$*$	$/$	$($	$)$	\$
S									acc
E			$push$	$push$				$push$	r0
A	r8	r8					r8		
T			r1	r1	$push$	$push$		r1	r1
B	$push$	$push$					$push$		
F			r5	r5	r5	r5		r5	r5

	n	i	$+$	$-$	$*$	$/$	$($	$)$	\$
n			r9	r9	r9	r9		r9	r9
i			r10	r10	r10	r10		r10	r10
$+$	r2	r2					r2		
$-$	r3	r3					r3		
$*$	r6	r6					r6		
$/$	r7	r7					r7		
$($	r4	r4					r4		
$)$			r11	r11	r11	r11		r11	r11
#	r4	r4					r4		

```
type
  TTypSymbolu = (S_ID, S_NUM, S_PLUS, S_MINUS,     // terminální symboly
    S_MUL, S_DIV, S_LPAR, S_RPAR, S_ENDOFFILE,
    S_NS, S_NE, S_NA, S_NT, S_NB, S_NF, S_HASH); // neterminální symboly
  ... // datové typy TZnak a TSymbol definované jako v předchozích sekcích

var
  konec:      boolean;      // indikátor ukončení výpočtu, proveden accept
  symbol:     TSymbol;      // aktuální symbol načtený z proměnné vstup
  znak:       TZnak;        // aktuální znak načtený ze zdrojového souboru
  vrchol_zas: TTypSymbolu;  // symbol na vrcholu zásobníku
  zasobnik:   TZasobnik;    // zásobník, prvky jsou zatím typu TTypSymbolu

procedure reduce(cislo_prav: integer);
begin
  case cislo_prav of
    0: begin                          // S → #E
      Vyjmi_ze_zasobniku;       // E
      Vyjmi_ze_zasobniku;       // #
      Pridej_do_zasobniku(S_NS);// také inicializuje vrchol_zasobniku
    end;
    1: begin                          // E → AT
      Vyjmi_ze_zasobniku;       // T
      Vyjmi_ze_zasobniku;       // A
      Pridej_do_zasobniku(S_NE);
    end;
    2: begin                          // A → E+
      Vyjmi_ze_zasobniku;       // +
      Vyjmi_ze_zasobniku;       // E
      Pridej_do_zasobniku(S_NA);
    end;
    3: begin                          // A → E-
      Vyjmi_ze_zasobniku;       // -
      Vyjmi_ze_zasobniku;       // E
      Pridej_do_zasobniku(S_NA);
    end;
```

```
    4: begin                          // A → ε
      Pridej_do_zasobniku(S_NA);
    end;
    ... // pro každé pravidlo gramatiky
        // zredukujeme v zásobníku pravou stranu pravidla na levou
  end;
  vystup(cislo_prav);
end;

procedure error(const hlaska: string);
begin
  konec := true;
  writeln('Chyba při syntaktické analýze na řádku ',znak.cislo,
    ', sloupci ',znak.pozice,': ',hlaska);
end;

procedure push;
begin
  Pridej_do_zasobniku(symbol.typ);
  Lex;         // lexikální analyzátor načte další symbol
end;

procedure accept;
begin
  Konec := true;
end;
procedure Init;
begin
  ...                          // inicializace vstupu a výstupu
  Vytvor_zasobnik;
  Pridej_do_zasobniku(S_HASH); // symbol konce zásobníku
  Lex;                         // načte symbol ze vstupu do vstupni_sym
  Konec := false;
end;

procedure Done;
begin
  Zlikviduj_zasobnik;        // uvolní paměť zabranou zásobníkem
  ...                        // uzavření vstupu a výstupu
end;

procedure Akce;
begin
  case vrchol_zas of
    S_NS: if symbol.typ = S_ENDOFFILE then accept
      else error('chybný symbol na vstupu - '+symbol.typ);
    S_NE: if (symbol.typ in [S_PLUS,S_MINUS,S_RPAR]) then push
      else if symbol.typ = S_ENDOFFILE then reduce(0)
      else error('chybný symbol na vstupu - '+symbol.typ);
```

```
    S_NA: if (symbol.typ in [S_ID,S_NUM,S_LPAR]) then reduce(8)
      else error('chybný symbol na vstupu - '+symbol.typ);
    S_NT: if (symbol.typ in [S_PLUS,S_MINUS,S_RPAR,S_ENDOFFILE])
      then reduce(1)
      else if (symbol.typ in [S_MUL,S_DIV]) then push
      else error('chybný symbol na vstupu - '+symbol.typ);
    S_NB: if (symbol.typ in [S_ID,S_NUM,S_LPAR]) then push
      else error('chybný symbol na vstupu - '+symbol.typ);
    S_NF: if (symbol.typ in [S_PLUS,S_MINUS,S_MUL,S_DIV,S_RPAR,S_ENDOFFILE])
      then reduce(5) else error('chybný symbol na vstupu - '+symbol.typ);
    S_NUM: if (symbol.typ in [S_PLUS,S_MINUS,S_MUL,S_DIV,S_RPAR,S_ENDOFFILE])
      then reduce(9)
      else error('chybný symbol na vstupu - '+symbol.typ);
    S_ID: if (symbol.typ in [S_PLUS,S_MINUS,S_MUL,S_DIV,S_RPAR,S_ENDOFFILE])
      then reduce(10)
      else error('chybný symbol na vstupu - '+symbol.typ);
    S_PLUS: if (symbol.typ in [S_ID,S_NUM,S_LPAR]) then reduce(2)
      else error('chybný symbol na vstupu - '+symbol.typ);
    S_MINUS: if (symbol.typ in [S_ID,S_NUM,S_LPAR]) then reduce(3)
      else error('chybný symbol na vstupu - '+symbol.typ);
    S_MUL: if (symbol.typ in [S_ID,S_NUM,S_LPAR]) then reduce(6)
      else error('chybný symbol na vstupu - '+symbol.typ);
    S_DIV: if (symbol.typ in [S_ID,S_NUM,S_LPAR]) then reduce(7)
      else error('chybný symbol na vstupu - '+symbol.typ);
    S_LPAR,S_HASH: if (symbol.typ in [S_ID,S_NUM,S_LPAR]) then reduce(4)
      else error('chybný symbol na vstupu - '+symbol.typ);
    S_RPAR: if (symbol.typ in [S_PLUS,S_MINUS,S_MUL,S_DIV,S_RPAR,S_ENDOFFILE])
      then reduce(11)
      else error('chybný symbol na vstupu - '+symbol.typ);
    else error('chybný symbol na vstupu - '+symbol.typ);
  end;
end;

procedure S_analyza;
begin
  Init;
  while (not Konec) do Akce;
  Done;
end;
```

Metoda přepisu rozkladové tabulky je u LR překladů z našeho hlediska výhodnější, protože „obrácená“ rekurze u rekurzivního sestupu (vlastně tady vzestupu) by se nám hůře programovala. Problémy se sémantikou narozdíl od uplatnění metody na LL jazyky nenastávají, jak zjistíme v dalších kapitolách.

Úkoly ke kapitole 3

1. 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.

   $S \to aABc \mid bB$
   $A \to aAA \mid aAbA \mid \varepsilon$
   $B \to bB \mid c$

2. Podle následující gramatiky vytvořte

   (a) množiny FIRST řetězců bA, $P\,;A$, $B < B$,
   (b) všechny množiny FOLLOW.

   $S \to DbAe.$
   $A \to P\,;A \mid \varepsilon$
   $B \to c \mid p$
   $D \to p\,;D \mid \varepsilon$
   $M \to B < B \mid B = B$
   $P \to iMtP \mid wV \mid rV \mid p = V$
   $V \to B + B \mid B - B \mid B * B$

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

3. Vypočtěte:

   (a) u první uvedené gramatiky množiny FIRST, FIRST_2 a FIRST_3 řetězců B, AB, Bb, BA,
   (b) u druhé uvedené gramatiky množiny FIRST a FIRST_2 řetězců dB, A, CA,
   (c) u obou gramatik všechny množiny FOLLOW a FOLLOW_2.

   $S \to xAB \mid yBA$
   $A \to axaX \mid bBa \mid \varepsilon$
   $B \to bybA \mid Bb \mid \varepsilon$

   $S \to AaaB \mid aS \mid \varepsilon$
   $A \to cA \mid bS \mid \varepsilon$
   $B \to CA \mid Bd$
   $C \to SdB \mid c \mid \varepsilon$

   U každé gramatiky porovnejte množiny FIRST_k a také množiny FOLLOW_k pro různá čísla k, všimněte si vztahu mezi nimi.

4. Podle následující gramatiky vytvořte

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

$S \to abA \mid cbBC$
$A \to cA \mid dB \mid a$
$B \to bBS \mid a$
$C \to cC \mid cBd \mid \varepsilon$

5. Ověřte, zda gramatika v úkolu 4 je $LL(1)$.
6. U každé z následujících gramatik zjistěte, zda je $LL(1)$ nebo silná $LL(2)$.

$S \to AbB \mid aBb$
$A \to abAA \mid c \mid \varepsilon$
$B \to daB \mid \varepsilon$

$S \to aABbCD \mid \varepsilon$
$A \to ASd \mid \varepsilon$
$B \to SAc \mid xC \mid \varepsilon$
$C \to Sy \mid Cz \mid \varepsilon$
$D \to aBD \mid \varepsilon$

$S \to acA \mid AaB$
$A \to \varepsilon \mid cAB$
$B \to bB \mid d$

7. 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.

$S \to aAB \mid bBS \mid \varepsilon$
$A \to cBS \mid \varepsilon$
$B \to dB \mid m$

8. Ověřte, zda gramatika v úkolu 2 je $LL(1)$. Pokud ne, transformujte ji na $LL(1)$. Podle gramatiky vytvořte derivaci jakéhokoliv slova w o délce alespoň 4.

   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.
9. Pokud je gramatika v úkolu 7 typu $LL(1)$ a máte rozkladovou tabulku, naprogramujte syntaktickou analýzu s použitím této tabulky, a to metodou přepisu rozkladové tabulky i metodou rekurzivního sestupu.
10. Ověřte, zda je následující gramatika silná $LL(2)$. Pokud ano, vytvořte rozkladovou tabulku a podle ní zpracujte jakékoliv slovo délky alespoň 4 generované gramatikou.

$S \to acA \mid AaB$
$A \to cAB \mid \varepsilon$
$B \to bB \mid d$

11. Zjistěte, zda je následující gramatika $LL(1)$. Pokud ne, zjistěte, zda je silná $LL(2)$.

$S \to AbB \mid aBb$
$A \to abAA \mid c \mid \varepsilon$
$B \to daB \mid \varepsilon$

12. Pozorně prostudujte rozkladové tabulky v kapitole 3.21 v příkladu 3.21 na straně 75. Proč v tabulce pro silnou $LL(2)$ gramatiku není žádný sloupec označen řetězcem začínajícím písmenem b, resp. proč je tento sloupec v $LL(1)$ tabulce prázdný?

13. U dané gramatiky vypočtěte všechny množiny FOLLOW, FOLLOW_2 a FOLLOW_3, včetně všech množin FIST_k, které budete potřebovat. Zjistěte, zda je tato gramatika silná $LL(k)$ pro některé $k = 1, 2, 3$, a pokud ano, sestrojte rozkladovou tabulku.

    $S \to AB$
    $A \to aAb \mid ab$
    $B \to cBd \mid cd$

    Nápověda pro kontrolu: v sestrojené rozkladové tabulce (resp. její horní části s řádky ohodnocenými neterminály) budou sloupce ohodnoceny řetězci, které budou mít všechny stejnou délku (samozřejmě k), žádný z nich nebude kratší. Součástí některého řetězce může být také symbol \$.

14. Otestujte následující gramatiku, zda je silná $LR(1)$ (nezapomeňte gramatiku předem rozšířit). Pokud ano, vytvořte rozkladovou tabulku a podle tabulky zpracujte některé ze slov patřících do jazyka generovaného gramatikou.

    $S \to aABb \mid \varepsilon$
    $A \to Ac \mid b$
    $B \to Bd \mid g$

15. Pokud je gramatika z předchozího příkladu typu silná $LR(1)$ a tedy máte rozkladovou tabulku, proveďte její implementaci metodou přepisu rozkladové tabulky.

KAPITOLA 4

Sémantická analýza

Jakmile syntaktický analyzátor najde určitou konstrukci symbolů, tedy frázi, je třeba této konstrukci přiřadit význam, přidat sémantiku. Sémantická analýza se stará především o to, aby se správně zacházelo s proměnnými a datovými typy (například lze vyžadovat, aby proměnná byla předem deklarována, provádí se typová kontrola apod.).

V této kapitole se podíváme na vytváření struktur a kódu potřebných pro sémantickou analýzu.

Vstup:	*konstrukce symbolů vytvořená syntaktickým analyzátorem*
Výstup:	*cílový kód, některá z forem interního kódu nebo interpretace*
Sémantické chyby:	*souvisí s významem symbolů a skupin symbolů, např. použití nedeklarované proměnné v kódu, nekompatibilní datový typ proměnné ve výrazu či argumentu funkce*

Tabulka 4.1: Vlastnosti sémantické analýzy

4.1 Tabulka symbolů

Do tabulky symbolů (tabulky objektů) ukládáme postupně všechny objekty pojmenované identifikátory (těmi, které nejsou klíčovými slovy) – obvykle jde o proměnné nebo konstanty, uživatelské datové typy, funkce, procedury, návěští apod., na které při analýze kódu narazíme.

Pojem *objekt* zde budeme chápat obecněji než je obvyklé v teorii programování. Bude to prostě jakýkoliv identifikátor, který není klíčovým slovem a lexikální analýza ho proto ještě neodlišila od jiných identifikátorů.

4.1.1 Význam tabulky symbolů

Do tabulky symbolů zapisujeme obvykle název, typ, adresu případně počáteční hodnotu objektu, počet a typ parametrů funkce a další informace potřebné při překladu, ale také při provádění programu.

Příklad 4.1
Tabulka symbolů může vypadat takto:

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

Tabulka 4.2: Tabulka symbolů

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

Tvar adresy závisí na struktuře interního kódu, který vytváří syntaktický a sémantický analyzátor. Může to být relativní adresa v paměti, index (pořadové číslo) fráze, ... Ovšem třeba u interpretačního překladače, kdy je tabulka symbolů přímo využívána pro uschování hodnot objektů, vůbec adresy nepotřebujeme.

U každého typu objektu potřebujeme uchovávat různé druhy informací. Například u *proměnné* je to název, adresa, datový typ, velikost potřebné paměti apod., u *funkce* název, adresa, návratový typ, počet a typ jednotlivých parametrů (můžou být odkazy na jiné řádky tabulky), příp. zda jsou volány hodnotou nebo odkazem (jestliže jsou volány odkazem, musí sémantický analyzátor navíc ošetřit, aby ve volání funkce byly jako skutečné parametry použity pouze názvy proměnných a nikoli například výrazy nebo konstantní hodnoty), pro *třídu* evidujeme název, předka (odkaz na jiný řádek tabulky), vlastnosti, vnitřní proměnné apod., u dalších typů objektů to budou opět jiné údaje.

Řádky tabulky mohou být navzájem závislé (jeden uživatelský datový typ může využívat deklaraci již dříve uvedeného, popř. proměnná je typu deklarovaného dříve, ...), v běžných případech se však nesmí jednat o kruhovou závislost (výjimkou jsou dopředné definice).

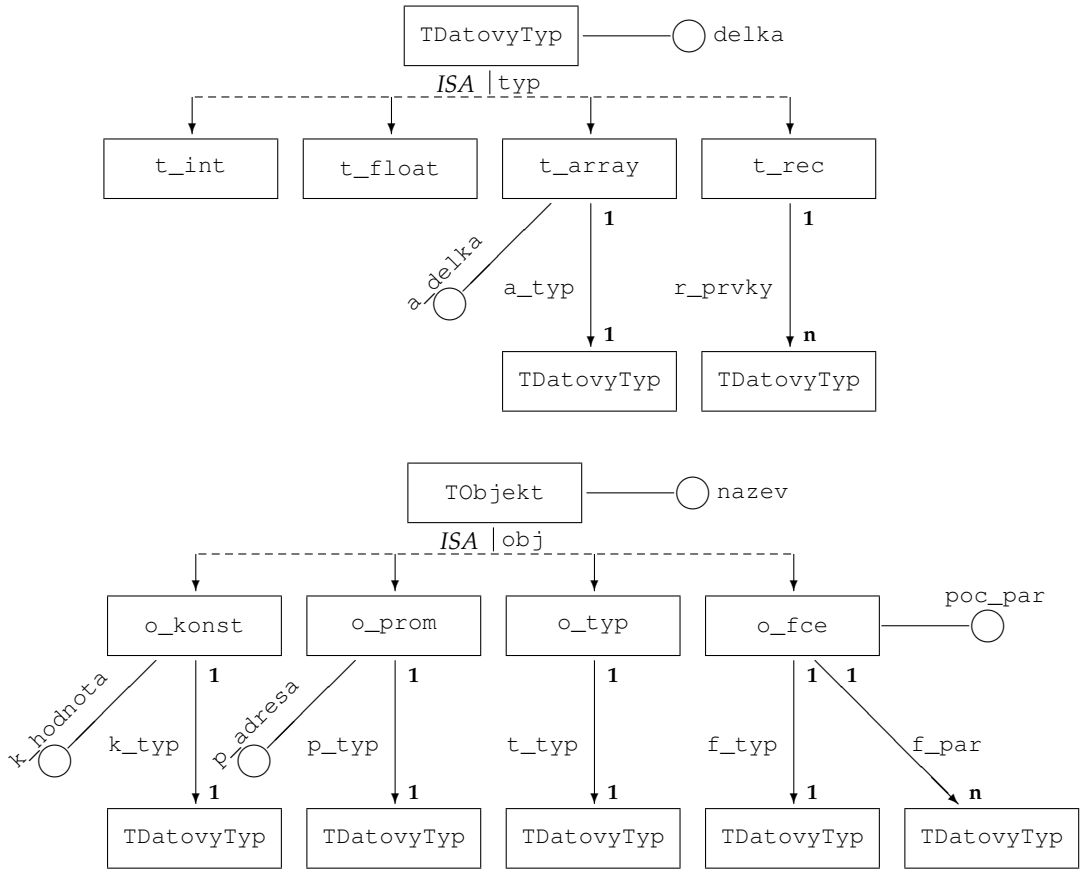
Tabulka symbolů nám slouží k mnoha účelům. Využívá ji zejména sémantický analyzátor (kontroluje, zda proměnná použitá v kódu je deklarovaná a zda její datový typ odpovídá

jejímu použití, jestli u funkce souhlasí počet a typ argumentů, atd.), používá se také u generování cílového kódu (překladač potřebuje informaci o tom, kolik místa v paměti má vyhradit pro jednotlivé symboly) a interpretace. Při interpretaci obvykle není nutné uchovávat informaci o adrese, samotná tabulka symbolů může sloužit jako úschovna symbolů (proměnných, objektů), se kterou pak neustále pracujeme.

4.1.2 Implementace

Pro implementaci tabulky symbolů používáme metody obdobné metodám v databázích. Vytvoříme E-R diagramy a ty pak naprogramujeme pomocí variantních záznamů (v jazyce C struktur s unionem), pointerů a dynamických struktur.

Na obrázku 4.1 vidíme E-R diagramy datového typu a běžného objektu pro jednoduchý jazyk zpracovávaný kompilačním překladačem. Jak vidíme, v tabulce symbolů jsou uloženy konstanty, číselné proměnné, uživatelsky definované datové typy (pole a záznam) a funkce. Způsob naprogramování najdeme v příkladu 4.2.



Obrázek 4.1: E-R diagramy pro jednoduchý kompilační překladač

Příklad 4.2

Naprogramujeme jednoduchou reprezentaci tabulky symbolů jazyka s E-R diagramy podle obrázku 4.1.

```
type
  TTypObjektu  = (o_konst, o_prom, o_typ, o_fce);
  TTypHodnoty = (t_int, t_float, t_array, t_rec);

  PDatovyTyp = ↑TDatovyTyp;
  TDatovyTyp = record
    delka: integer;
    case typ: TTypHodnoty of
      t_int:   ();
      t_float: ();
      t_array: (a_typ: PDatovyTyp; a_delka: integer);
      t_rec:   (r_prvky: PDatovyTyp); // dynamické pole prvků
  end;

  PObjekt = ↑TObjekt;
  TObjekt = record
    dalsi: PObjekt;
    nazev: string[MaxDelkaNazvu];
    case obj: TTypObjektu of
      o_konst: (k_hodnota: THodnota; k_typ: TDatovyTyp);
      o_prom:  (p_adresa: integer; p_typ: TDatovyTyp);
      o_typ:   (t_typ: TDatovyTyp);
      o_fce:   (f_typ: TDatovyTyp; f_par: ↑TDatovyTyp; poc_par: byte);
  end;

var TabulkaSymbolu: ↑TObjekt;
```

Příklad 4.3

Kód z příkladu 4.2 upravíme pro interpretační překladač tak, aby bylo možné do tabulky symbolů ukládat i hodnoty proměnných.

```
type
  TTypObjektu  = (o_konst, o_prom, o_typ, o_fce);
  TTypHodnoty = (t_int, t_float, t_array, t_rec);

  PHodnota = ↑THodnota;
  THodnota = record
    case typ: TTypHodnoty of
      t_int:   (i: integer);
      t_float: (f: float);
      t_array: (a_typ: TTyp; a_delka: integer; a_prvky: PHodnota);
      t_rec:   (r_prvky: PHodnota);
  end;
```

```
  PObjekt = ↑TObjekt;
  TObjekt = record
    dalsi: PObjekt;
    nazev: string[MaxDelkaNazvu];
    hodnota: THodnota;                // u funkce návratová hodnota
    case obj: TTypObjektu of
      o_konst: ();
      o_prom:  ();
      o_typ:   ();
      o_fce:   (f_par: PObjekt; poc_par: integer);
  end;

var
  TabulkaSymbolu: ↑TObjekt;
```

Rozdíl je především v tom, že kromě datového typu evidujeme také hodnoty proměnných. Prvky tabulky jsou typu `TObjekt`. V tomto záznamu se přímo dostaneme k typu (vnitřní proměnná `obj`) a hodnotě proměnné, u funkce navíc máme v záznamu počet parametrů a jednotlivé parametry (dynamický seznam `f_par`).

Celou tabulku realizujeme tak, aby bylo snadné v ní vyhledávat. Může to být statický nebo dynamický seznam řádků, binární strom (velmi výhodný pro vyhledávání), dynamické pole záznamů, řídká tabulka nebo jiné struktury známé také z teorie databází, případně použijeme hashování. Velmi často se používá možnost binárního vyhledávání nebo jiné vyhledávací metody.

Otázkou je, jak vlastně řadit jednotlivé objekty v tabulce. Důležitým kritériem je rychlost vyhledávání, protože k tabulce symbolů přistupuje zejména sémantický analyzátor velmi často. U jednodušších jazyků je možné tabulku automaticky řadit podle abecedy, u složitějších jazyků to však není moc vhodné (odporuje to následujícímu kritériu). Proto problém můžeme řešit například indexací, kdy zároveň s tabulkou vytváříme indexový seznam (příp. soubor), ve kterém jsou odkazy na objekty seřazené podle abecedy.

Dalším kritériem při tomto rozhodování je vzájemná závislost obsažených objektů – uživatelsky definovaný datový typ definujeme s použitím již definovaných, proměnnou deklarujeme vždy s použitím existujícího datového typu, příp. pokud jazyk dovoluje proměnnou inicializovat (nebo se provádí automatická inicializace), lze využít hodnotu již deklarovaných proměnných, atd. Pak nezbývá než objekty řadit do tabulky tak, jak jsou postupně načítány.

Určitou komplikací je věc, kterou umožňují některé existující jazyky, a to možnost vytvoření datového typu na základě jiného datového typu, který ještě nebyl vytvořen (viz dynamické struktury v Pascalu). Pokud je jazyk takto definován, pak dynamická konstrukce tabulky symbolů netvoří strom, mohou se v ní objevit i cykly, a proto je nutné upravit algoritmy pro procházení tabulkou. Za určitých okolností je dokonce nutné provést

lexikální analýzu předem, a to postupně v alespoň dvou průchodech, aby byly zachyceny všechny identifikátory, na které se lze odvolávat.

4.1.3 Tabulka symbolů vytvářená lexikálním analyzátorem

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

Jak již víme z předchozích kapitol, symboly můžeme reprezentovat (variantním) záznamem, ve kterém je údaj o typu symbolu a také hodnoty atributů. Pro identifikátory (proměnné, názvy funkcí apod.) jsme dosud v tomto záznamu používali atribut typu řetězec, ale máme také jinou možnost, která nejen šetří místo v paměti rezervované pro názvy proměnných, ale také redukuje počet prohledávání tabulky symbolů. Místo řetězcového atributu použijeme v záznamu symbolu pouze ukazatel (pointer) do tabulky symbolů.

Příklad 4.4

Pokud máme tabulku symbolů naprogramovanou podle příkladu 4.3, můžeme upravit tvar záznamu pro uchování symbolů (jde především o identifikátory).

Místo řetězce pro název proměnné zavedeme ukazatele do tabulky symbolů. Ukládané typy atributů rozlišíme podle typu symbolu, uvedeme pouze ty typy, které vyžadují atributy.

```
type
  TSymbol = record
  case typ: TTypSymbolu of
    S_NUMINT: (i: integer);
    S_NUMFLT: (f: float);
    S_ID:     (hodn: ↑TObjekt);
  end;
```

V tomto případě tabulku symbolů vytváří lexikální analyzátor, protože také v této fázi je třeba znát pozici identifikátoru v tabulce (při směrování ukazatelů).

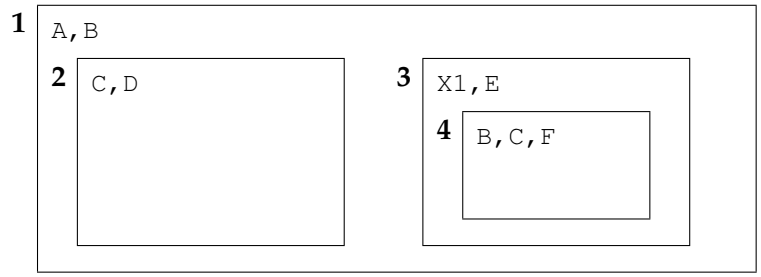
4.1.4 Tabulka symbolů pro program s blokovou strukturou

Speciální implementaci vyžaduje tabulka symbolů pro jazyk s blokovou strukturou, jako je třeba PASCAL. Rozlišují se zde lokální a globální objekty a přístupnost lokálních objektů bývá omezená.

Například máme tuto strukturu bloků:

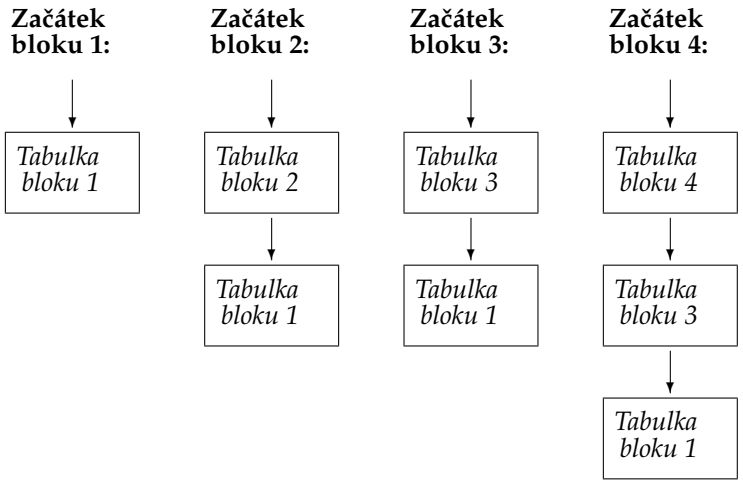
- V hlavním bloku jsou deklarovány proměnné `A, B`.
- Uvnitř bloku je volán blok č. 2, ve kterém jsou proměnné `C, D`.
- Po ukončení bloku č. 2 je volán blok č. 3 s proměnnými `X1, E`.
- Uvnitř bloku č. 3 je volán blok č. 4 s proměnnými `B, C, F`.

Popsaná struktura je znázorněna na obrázku 4.2.



Obrázek 4.2: Tabulka symbolů pro jazyk s blokovou strukturou.

Každá proměnná je viditelná v tom bloku, ve kterém je deklarována, a také ve všech blocích vnořených, tedy například proměnnou `A` lze používat ve všech blocích, zatímco proměnnou `D` pouze v bloku č. 2 a `F` pouze v bloku č. 4.



Obrázek 4.3: Tabulka symbolů pro jazyk s blokovou strukturou

Když v určitém bloku použijeme proměnnou, hledáme informace o ní nejdřív v tom bloku, ve kterém se nacházíme. Při neúspěchu se posouváme do nadřízeného bloku (podle nákresu směrem dolů) a tak postupujeme, dokud proměnnou nenajdeme. Pokud neuspě-

jeme ani v hlavním bloku, znamená to, že byla použita proměnná, která není deklarována, jde o sémantickou chybu.

Když v bloku 2 použijeme proměnnou `B`, hledáme ji (informace o ní) nejdřív v bloku 2, kde ovšem není, a tak postoupíme do bloku 1, kde už ji nalezneme. Když tuto proměnnou použijeme v bloku 4, nemusíme postupovat do nadřízeného bloku a použijeme přímo definici z bloku 4. Jestliže je proměnná deklarována ve více blocích, které jsou vzájemně vnořeny, bereme vždy nejbližší deklaraci (nejblíže vrcholu zásobníku).

Každý blok má svou vlastní tabulku. V bloku 3 už nepotřebujeme informace o objektech bloku 2, proto zrušíme vyhledávací vazbu na tabulku bloku 2 (ale v paměti může zůstat). Během překladu jednotlivých bloků je možné tabulky zřetězit do datové struktury „zásobník“ podle obrázku 4.3.

S celou strukturou se pracuje jako s klasickým zásobníkem. Každá z tabulek má svou vlastní organizaci a je z ní přístupná nadřízená tabulka. „Aktivní“ tabulka je na vrcholu zásobníku, kde také začínáme prohledávat. Při vyhodnocení konce bloku se aktivní tabulka ze zásobníku odstraní, aktivní se stane k ní nadřízená tabulka. Tabulka hlavního bloku (blok 1) zůstává v zásobníku až do konce vyhodnocování programu, je odstraněna až jako poslední po vyhodnocení celého programu.

Implementace celé struktury závisí především na implementaci tabulek jednotlivých bloků. U jednoduchého jazyka můžeme všechny symboly prostě ukládat do jediného zásobníku a v jiném, pomocném, zásobníku evidujeme pomocí ukazatelů, kde v hlavním zásobníku začíná či končí která „podtabulka“ (v našem příkladu by nejdřív v pomocném zásobníku byl jediný ukazatel, a to na začátek symbolů prvního bloku – začátek celého hlavního zásobníku, po příchodu do bloku 2 přidáme do pomocného zásobníku ukazatel na první objekt, který do hlavního zásobníku přidáme z bloku 2, po opuštění bloku 2 tento ukazatel z pomocného zásobníku odstraníme, . . .).

U složitějších jazyků můžeme opět využít zásobník s tím, že pro tabulky bloků použijeme samostatné datové struktury vhodné pro vyhledávání (binární strom, hashovací tabulka, . . .), nebo celou strukturu „obrátíme“: v binárním stromě nebo hashovací tabulce máme uschovány objekty podle jména a u každého udržujeme seznam bloků, ve kterých jsou deklarovány s tím, že přístupný je pouze záznam o posledním bloku, ve kterém je objekt v daném okamžiku použit. Při odchodu z určitého bloku je nutné vždy projít tabulku a odstranit všechny aktivní záznamy, které se ho týkají (procházíme ty objekty, které byly v bloku platné).

U jazyků s blokovou strukturou syntaxe obvykle nepoužíváme pro vstupní symboly typu identifikátoru ukazatel do tabulky symbolů (jak bylo popsáno v sekci 4.1.3). Pokud se však pro tuto reprezentaci rozhodneme, musíme zajistit, aby lexikální analyzátor byl ve stejném průchodu jako syntaktický, aby bylo možné vytvářet ukazatele do správné tabulky pro daný blok.

4.2 Intermediální kód

Většina překladačů po syntaktické a sémantické analýze negeneruje přímo program v cílovém jazyce, ale vytváří program v intermediálním kódu, který lze snadněji optimalizovat, a teprve optimalizovaný program převede do cílového jazyka. Tuto formu kódu lze také použít pro interpretaci. Interním kódem, který generuje sémantický analyzátor, bývá právě intermediální kód.

Definice 4.1 (Intermediální kód) *Intermediální kód je výstupní kód sémantické analýzy nebo některého optimalizačního průchodu. Jde o mezikód, který je dále zpracováván – interpretován, optimalizován nebo přeložen do cílového kódu. Má různé varianty s ohledem na jeho další použití.*

Hlavní rozdíl mezi intermediálním a cílovým kódem je v tom, že v intermediálním kódu není ještě stanovena adresace operandů ani použití registrů pro výpočty, a také se snadněji zpracovává.

Na intermediální kód se kladou různé požadavky podle toho, zda je překladač interpretační nebo kompilační.

- Intermediální kód pro kompilační překladač musí být snadno optimalizovatelný (například grafovými algoritmy) a snadno přepsatelný do strojového kódu nebo assembleru.
- Intermediální kód pro interpretační překladač musí být snadno interpretovatelný, pokud možno bez dalších větších úprav.

Nejčastěji se používají tyto formy (případně jejich modifikace nebo kombinace):

- 3-adresový kód,
- sémantický strom (ohodnocený syntaktický strom),
- postfixový tvar (polská notace).

U interpretačních překladačů skutečný intermediální kód ani nemusí být vytvářen a nebo souvisí pouze se syntaktickou strukturou programu.

4.2.1 3-adresový kód

Příkaz v 3-adresovém kódu má tvar

- základní: (operátor, argument$_1$, argument$_2$, výsledek),
- zhuštěný: (operátor, argument$_1$, argument$_2$).

Příklad 4.5

Převedeme následující výraz do základní a zhuštěné formy 3-adresového kódu:

```
A := (-B) * (C + D)
```

Základní:

Upravíme:
`T1 := -B`
`T2 := C + D`
`T3 := T1 * T2`
`A := T3`

Op	*Arg1*	*Arg2*	*Výsl*
uminus	B		T1
+	C	D	T2
*	T1	T2	T3
:=	T3		A

Zhuštěný:

	Op	*Arg1*	*Arg2*
①	*uminus*	B	
②	+	C	D
③	*	①	②
④	:=	A	③

Všimněme si také rozlišení běžného binárního operátoru „−“ a unárního operátoru obvykle značeného stejně, zde je nutné tyto dva případy rozlišit (unární operátor je zde označen *uminus*).

Na příkladu 4.5 vidíme jistou podobnost především s assemblerem (případně také s prefixovým kódováním).

Například druhý řádek lze v assembleru procesorů Intel napsat těmito třemi instrukcemi (předpokládejme, že pro aritmetické operace máme určen registr `AX`[1]):

```
MOV AX, C     ; hodnota proměnné C se přesune do registru AX      AX = C
ADD AX, D     ; hodnota proměnné D se přičte do registru AX       AX = AX+D
MOV T2, AX    ; hodnota uložená v AX se přesune do proměnné T2    T2 = AX
```

Význam uvedených instrukcí je zřejmý – `MOV` je zkratka z „move“, slouží k přesouvání hodnot mezi paměťovými místy, obvyklým požadavkem je, aby alespoň jedním z parametrů byl některý registr, a `ADD` je instrukce pro sčítání.

Stejným způsobem se dají převést i ostatní řádky, nesmíme ovšem zapomenout na zařazení dočasných proměnných do tabulky symbolů.

Nyní se podíváme na možnost převodu složitějších příkazů do 3-adresového kódu. Protože tento typ intermediálního kódu se využívá především u kompilačních překladačů, „vypůjčíme si“ některé typy operátorů z assembleru, který bývá v těchto případech často cílovým kódem překladu, konkrétně

- návěští,
- instrukci `CMP` pro porovnání, její výsledek je využit následující instrukcí,
- instrukci `JG`, která znamená odskok na dané návěští, pokud při předchozím porovnání je mezi dvěma čísly vztah „>“ (jump if greater),
- instrukci `JL`, která znamená odskok na dané návěští, pokud při předchozím porovnání je mezi dvěma čísly vztah „<“ (jump if less),

[1] Registry jsou paměťová místa přímo v procesoru. Protože k nim má procesor přímý přístup, práce s nimi je mnohem rychlejší než práce s jinými typy paměti, proto se hodně používají právě v aritmetických operacích. Některé instrukce přímo vyžadují alespoň jeden z registrů jako svůj parametr, některé dokonce používají určité registry jako implicitní parametry (např. instrukce `MUL par` pro násobení provádí akci `AX = AX * par`, tedy implicitním parametrem je registr `AX`).

- případně další podobné instrukce, například `JMP` pro odskok na dané návěští (bezpodmínečné, nic se netestuje), `JGE` – odskok na návěští v případě, že při předchozím porovnání je vztah mezi operandy „$\geq$“ (jump if greater or equal), atd.

V tabulce budeme mít o jeden sloupec víc – pro návěští.

Příklad 4.6

Zpracujeme do 3-adresového kódu výraz `for i := x to y do v := v + x`

Návěští	*Op*	*Arg1*	*Arg2*	*Výsl*
	:=	i	x	i
	CMP	i	y	
	JG	konec_for		
zacatek_for:	+	v	x	T1
	:=	v	T1	v
	INC	i		
	CMP	y	i	
	JL	zacatek_for		
konec_for:				

Jak vidíme, vždy se nejdřív provede porovnání příkazem `CMP` a následnou reakcí je příkaz skoku v případě, že porovnání splní určitou podmínku (například pro `JG` se skok na dané návěští `konec_for` provede, pokud u porovnání první operand byl větší než druhý a tedy nelze cyklus provést ani jednou).

Ve skutečném kódu musí být samozřejmě jednotlivá návěští pro různá použití tohoto příkazu odlišena (příkaz `for` může být přece v programu použit vícekrát), například k označení návěští můžeme přidat pořadové číslo nebo číslo řádku zdrojového kódu.

3-adresový kód má tyto vlastnosti:

- dobře se optimalizuje (i když ne grafovými algoritmy),
- dobře se přepisuje na assembler i strojový kód,
- hůře se přímo interpretuje,
- je výhodný pro kompilační překladač, nevhodný pro interpretaci.

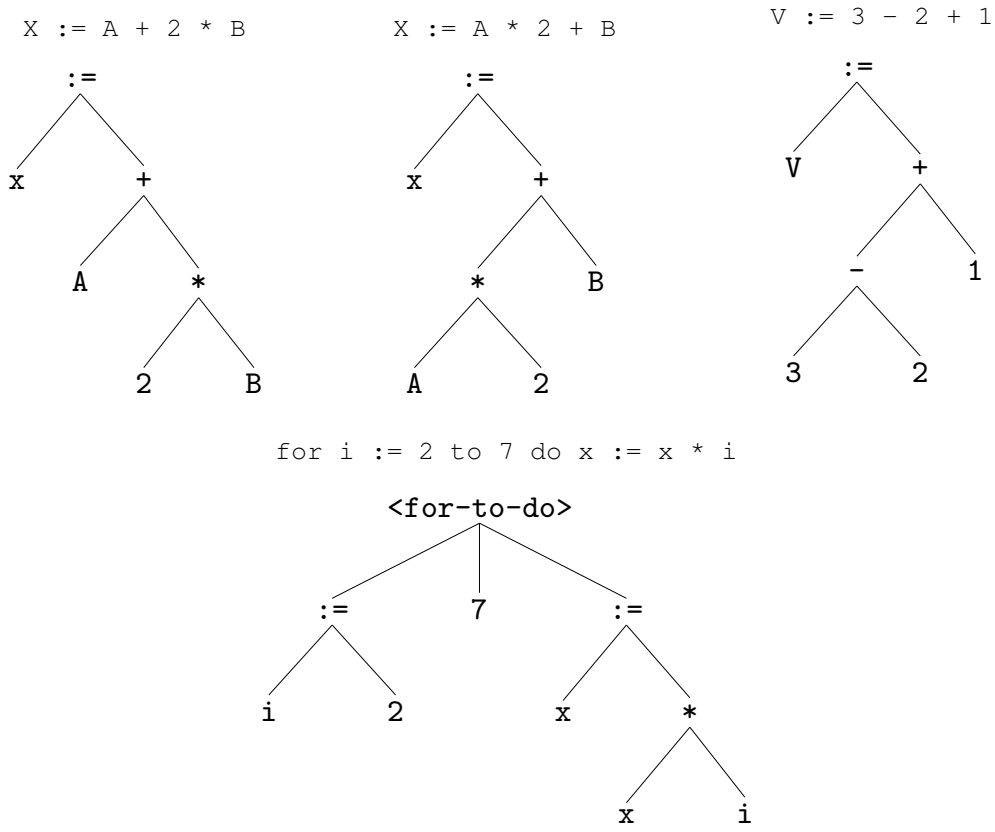
4.2.2 Sémantický strom

Sémantický (nebo také ohodnocený syntaktický) strom získáme úpravou derivačního stromu. Měly by zde zůstat pouze uzly obsahující terminální symboly, vnitřní uzly jsou ohodnoceny operátory, v listech pak najdeme operandy (například proměnné, čísla).

Pokud máme vytvořen derivační strom a v jeho listech (obsahujících terminální symboly) jsou uloženy i sémantické hodnoty (tj. atributy symbolů), můžeme vytvořit sémantický strom ekvivalentními úpravami grafu.

Tvar stromu ukážeme na příkladech.

Příklad 4.7



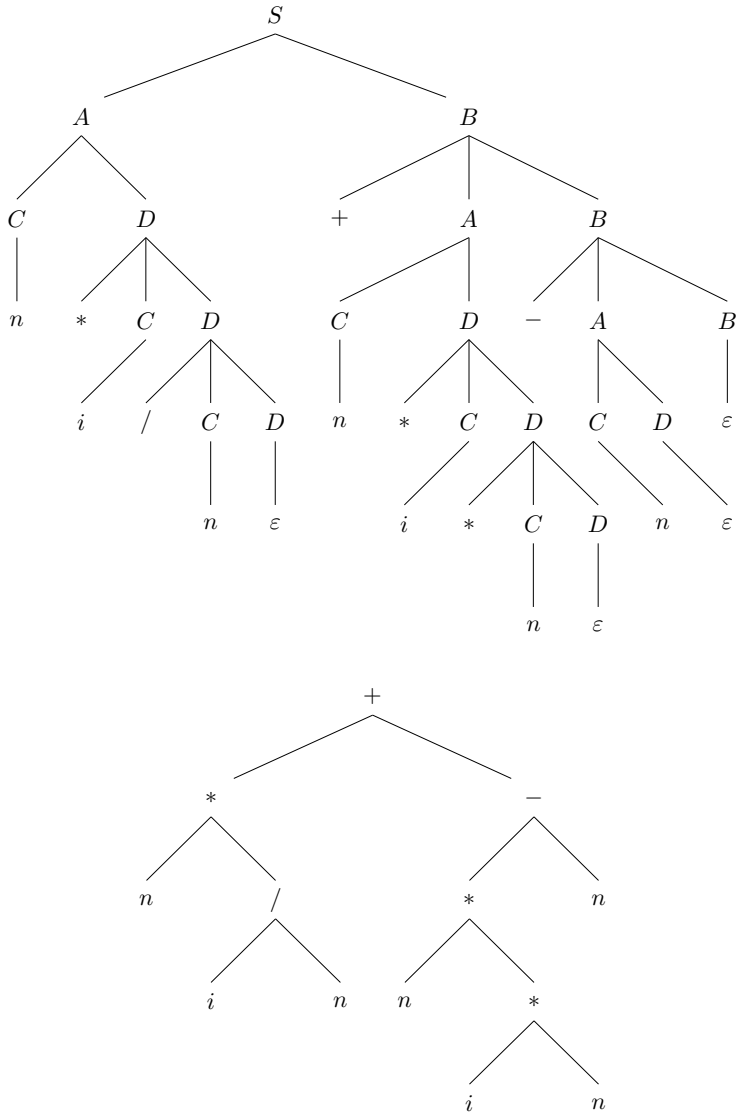
V příkladu 4.7 vidíme sémantické stromy přiřazovacího příkazu a příkazu typu `for`. Pro jiné typy příkazů by byl strom obdobný. Protože program obvykle chápeme jako posloupnost příkazů, tento typ intermediálního kódu pro celý program by měl tvar (lineárního) seznamu těchto dílčích sémantických stromů (včetně bloků příkazů), případně do sebe vnořených (složené příkazy). posloupnost příkazů je reprezentovaná seznamem stromů, struktura může být rekurzívně vnořovaná.

U vhodně sestavené $LL(1)$ gramatiky není problém vytvořit z derivačního stromu sémantický strom. Například pro běžnou gramatiku matematických výrazů je postup tento:

- odstraníme větve s ε-listy (odspodu přes neterminály k prvnímu větvení),
- všechny operátory posuneme o dvě úrovně nahoru po větvích,

- terminály, které nejsou operátory ani závorkami (tj. čísla, identifikátory apod.), posuneme o jednu nebo více úrovní nahoru po větvích (přes všechny neterminály k prvnímu větvení),
- levé závorky taktéž posuneme o úroveň nahoru, pravé závorky necháme na místě,
- prázdné větve (případně obsahující jen neterminály) odstraníme.

Na obrázku 4.4 je ukázka derivačního stromu a podle něho vytvořeného sémantického stromu.



Obrázek 4.4: Vytvoření sémantického stromu

Sémantický strom má tyto vlastnosti:

- výborně se přímo interpretuje (můžeme reprezentovat dynamickým stromem),
- sémantický strom lze propojit s grafickým editorem, který zprostředkuje práci s dynamickou strukturou,
- určité omezené možnosti optimalizace grafovými algoritmy,
- špatně se přepisuje na assembler nebo strojový kód, proto je vhodný spíše pro interpretaci.

4.2.3 Postfixový tvar

Budeme předpokládat, že pojmy infixový zápis, postfixový zápis a prefixový zápis jsou již čtenáři známy. Při převodu výrazu či jiného příkazu do postfixu v každém podvýrazu vždy přesouváme operátor za jeho operandy.

Příklad 4.8

Připomeneme převod běžného matematického výrazu do postfixového tvaru.

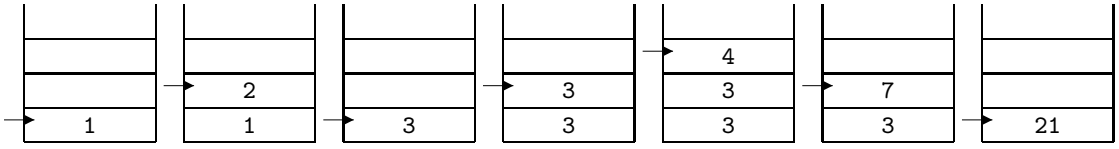
Infixový tvar: `Vysl := A * (X - 2 + Y) / 3`
Postfixový tvar: `Vysl A X 2 - Y + * 3 / :=`

Na příkladu 4.8 vidíme hlavní výhodu postfixového tvaru výrazu – struktura výrazu je zachycena bez nutnosti použití závorek. Samotný převod do postfixu můžeme provést například překladovou gramatikou, touto problematikou se budeme zabývat v kapitole 5.2.

Příklad 4.9

Následující nákresy ilustrují postup zpracování výrazu převedeného do postfixu pomocí jediného zásobníku:

`(1 + 2) * (3 + 4)` $\Longrightarrow$ `1 2 + 3 4 + *`



Čísla jednoduše ukládáme do zásobníku, a když v posloupnosti symbolů narazíme na operátor, vyjmeme postupně ze zásobníku požadovaný počet položek (například u binárního operátoru dvě položky) a operátor na ně uplatníme (dáváme pozor na pořadí).

Poznámka: Je třeba mít na paměti jednu důležitou věc: operátory se stejnou prioritou (třeba + a −) se vyhodnocují zleva doprava. Kdybychom provedli opačné vyhodnocení, vyšel by v mnoha případech nesprávný výsledek (vyzkoušejte na výrazu `3 - 2 + 1`).

Převod běžného matematického výrazu do postfixu zvládne každý, ovšem s jinými typy příkazů je to už horší. Pokud chceme použít tento druh intermediálního kódu pro všechny typy příkazů, můžeme si „vypůjčit" prostředky z jazyka, do kterého chceme zdrojový kód přeložit. Obvykle se pro tyto účely používá assembler, především odskoky na návěští, jak je patrné z příkladu 4.10.

Příklad 4.10

Rozhodovací příkaz převedeme do postfixového tvaru.

```
IF a + b <> 0 THEN x := 10 ELSE x := a + b
```

Předupraveno:

```
     a + b
     JZ @T
     x := 10      ; THEN
     JMP @K
@T:  x := a + b   ; ELSE
@K:               ; konec IF
```

Další úprava:

```
     a b +
     @T JZ
     x 10 :=      ; THEN
     @K JMP
@T:  x a b + :=   ; ELSE
@K:               ; konec IF
```

Po převodu dostaneme posloupnost

```
a b +   @T JZ   x 10 :=   @K JMP   @T: x a b + :=   @K:
```

Postfixový tvar má tyto vlastnosti:

- dá se optimalizovat, i když s obtížemi,
- dobře se přepisuje na assembler i strojový kód,
- dobře se interpretuje s použitím jediného zásobníku,
- vhodný pro interpretační i kompilační překladač.

4.3 Typová kontrola a přetypování

Každý operátor vyžaduje operandy určitého typu. Například pro operátor + mohou platit tyto předpisy:

- oba operandy jsou celá čísla, pak výsledek je celé číslo,
- oba operandy jsou reálná čísla, pak výsledek je reálné číslo,
- oba operandy jsou znaky, pak výsledek je řetězec (zřetězení těchto dvou znaků),
- oba operandy jsou řetězce, pak výsledek je řetězec (zřetězení těchto dvou řetězců).

Případy, kdy nemůžeme postupovat podle žádného z těchto předpisů, musíme ošetřit implicitním přetypováním nebo, pokud to není možné, hlásit chybu. Například pokud je první operand reálné číslo a druhý celé číslo, musíme druhý operand přetypovat na reálné číslo. Pokud je to možné (u konstant), může být toto přetypování provedeno už při překladu v rámci optimalizace.

Typová kontrola se vztahuje na kontroly datových typů operandů aritmetických a relačních operátorů, ale také na kontroly parametrů funkcí a prakticky tvaru všech příkazů, které obsahují proměnné prvky. Pro implicitní přetypování si v jazyce stanovíme posloupnost datových typů podle jejich priority, kterou při samotném přetypování používáme takto: jestliže mají být operandy stejného typu a není tomu tak, podle posloupnosti zjistíme, který z nich je typu s menší prioritou a ten přetypujeme. Například pro jazyk C je posloupnost pro čísla stanovena takto:

int — unsigned int — long — unsigned long — float — double — long double

Nejmenší prioritu má datový typ *int*, tedy kdyby u operátoru + jeden operand byl typu *long* a druhý *double*, první je přetypován na *double*. Operandy typu *char* a *short int* jsou při výpočtu výrazu vždy konvertovány na *int*, operandy typu *unsigned char* a *unsigned short* konverzí na *int* procházejí jen tehdy, když se vejdou do datového typu, tj. když nedojde k přetečení.

V moderních programovacích jazycích se setkáváme s pojmy *přetěžování operátorů* a *polymorfismus*. Způsob vyhodnocování a přetypování je dán množinou předpisů pro operátor a posloupností datových typů používanou při přetypování. Jestliže je možné operátory přetížit, pak již nestačí mít statickou posloupnost určující priority operandů a statickou množinu předpisů, tuto posloupnost musíme dynamicky měnit nebo rozšiřovat množinu předpisů.

Příklad 4.11

Navrhujeme programovací jazyk, který dovoluje definování vlastních datových typů a přetěžování operátorů tak, aby mohly být na tyto uživatelsky definované datové typy uplatňovány. Uživatel definuje vlastní datový typ `TComplex` pro komplexní čísla jako záznam dvou racionálních čísel (`float`). První z čísel je reálná část komplexního čísla, druhá imaginární část. Uživatel chce počítat s komplexními čísly s použitím operátorů +, −, atd., proto je nutné tyto operátory přetížit. Pro přetěžování binárních operátorů určíme funkce ve tvaru

```
operator (const op, par1: type1, par2: type2) → type3
```

Uvnitř funkce pak uživatel určí, jaká hodnota se má v závislosti na hodnotě parametrů (tj. operandů) vrátit. Například pro sčítání uživatel vytvoří tyto funkce:

```
operator (+, par1: TComplex, par2: TComplex) → TComplex;
begin
  result.realna := par1.realna + par2.realna;
  result.imaginarni := par1.imaginarni + par2.imaginarni;
end;
```

```
operator (+, par1: TComplex, par2: float) → TComplex;
begin
  result.realna := par1.realna + par2;
  result.imaginarni := par1.imaginarni;
end;
```

A další, případně lze definovat přetypování (třeba typu `float` na `TComplex`).

Polymorfismus znamená v některých programovacích jazycích možnost volání funkcí s tímtéž názvem pro různé typy parametrů nebo dokonce i různý počet parametrů a vracející hodnoty různých datových typů. Zde je třeba ošetřit správné určení konkrétní funkce, která je volána.

Podle způsobu zacházení s datovými typy rozlišujeme jazyky silně a slabě typované.

Silně typované jazyky provádějí při každé operaci kontrolu datových typů. Důsledkem nesrovnalostí zjištěných při této kontrole bývá chybové hlášení nebo dynamické přetypování. Některé silně typované jazyky vyžadují uvádění datového typu v deklaracích, jiné nikoliv. Jinými slovy, pro každou operaci se vyžadují operandy konkrétního typu. Tyto jazyky jsou nazývány *typově bezpečné*, což ale neznamená, že v nich nelze dělat chyby. K silně typovaným jazykům patří například ADA, JAVA, C# a PYTHON.

Slabě typované jazyky neprovádějí tolik kontrol datových typů a obecně je možné operaci určenou pro data jednoho datového typu použít na data jiného datového typu, alespoň do určité míry. Toto chování může být zdrojem mnoha běhových chyb. K slabě typovaným jazykům patří například jazyk C (ale C++ patří spíše k silně typovaným jazykům) nebo PERL. V jazyce C je například možné za určitých okolností sečíst celé číslo a řetězec, například

```
printf("%d", (1 + "1"));
```

Netypové jazyky (tj. naprosto bez datových typů) ve skutečnosti téměř ani neexistují (i když za netypové můžeme považovat třeba jazyky pracující s jediným datovým typem), vždy jsou definovány alespoň vnitřní datové typy, třebaže se s nimi programátor téměř nesetkává. Do této skupiny bychom snad mohli zařadit některé skriptovací jazyky, jako třeba některé běžné textové shelly včetně Příkazového řádku Windows.

Existují programovací jazyky, které se při určitých nastaveních mohou chovat buď jako silně typové nebo jako slabě typové. Typický případ je jazyk VISUALBASIC.NET, který byl ve výchozím nastavení v prvních verzích slabě typový, v posledních verzích je silně typový – nastavuje se použitím kompilátorové direktivy *Option Explicit On* (silně typový, nyní výchozí nastavení) nebo *Option Explicit Off* (slabě typový).

4.4 Statická a dynamická sémantika

Sémantiku programovacího jazyka můžeme rozdělit na dvě části – statickou a dynamickou. Liší se především v tom, kdy příslušnou sémantickou kontrolu provádíme.

Statická sémantika popisuje pravidla deklarace a definice jednotlivých prvků jazyka (jak mají deklarace vypadat, zda jsou nutné apod.), významu příkazů a jiných jazykových konstrukcí, typu parametrů těchto konstrukcí apod., provádí statickou typovou kontrolu a základní práci s tabulkou symbolů. Statická sémantika se řeší při překladu programu.

Dynamická sémantika se týká především významu jednotlivých jazykových konstrukcí (co se má provést, když je v programu napsáno . . . , priorita operátorů, atd.). Dynamickou sémantiku řešíme přímo za běhu programu.

Dynamická sémantika je náročná zvláště u programovacích jazyků s rekurzivním voláním podprogramů. Je nutné vést evidenci zvlášť o všech voláních téhož podprogramu, což se řeší vytvářením aktivačních záznamů pro jednotlivá volání a jejich ukládáním do zásobníku. Touto problematikou vztaženou na interpretaci se budeme zabývat v kapitole 6.4.4.

U programovacích jazyků se provádějí statické a dynamické (run-time) typové kontroly. Jak plyne z předchozího textu, statické typové kontroly se provádějí při překladu, dynamické za běhu programu. Zde se dostáváme k dalšímu kritériu rozdělení programovacích jazyků.

Staticky typované jazyky vyžadují uvádění datového typu u každé deklarace, nelze deklarovat proměnnou, objekt nebo funkci bez zadání datového typu. Prakticky všechny typové kontroly jsou prováděny při překladu (tj. jsou statické), již během překladu má být u každé proměnné jasné, jakého je datového typu.

Tyto jazyky nabízejí možnost explicitního přetypování, které však v praxi obvykle slouží především k obcházení typových kontrol. Výhodou je především lepší možnost odladění typových chyb, nevýhodou menší pružnost jazyka a větší složitost některých programových konstrukcí a kód programů bývá výrazně delší (například v JAVĚ musí být definována alespoň jedna třída i pro jednoduchý program typu „Ahoj světe“). Z běhových chyb se vyskytuje především přetečení datového typu (programátor se do proměnné pokouší uložit hodnotu, která se tam nevejde).

Staticky typovanými jazyky jsou například JAVA, ADA nebo C a jeho pokračovatelé.

Dynamicky typované jazyky nevyžadují uvádění datových typů v deklaracích, mnohé dokonce nevyžadují ani samotné deklarace. V takovém případě je však obvykle první použití proměnné vázáno na formu l-hodnoty (tj. na levé straně přiřazovacího příkazu), datový typ a hodnota nově vytvořené proměnné se řídí výsledkem vyhodnoceného výrazu. Velká část typových kontrol (ne-li všechny) se provádí dynamicky za běhu programu (dynamická sémantika), běžně se provádí automatické přetypování a pro uživatele také není problém kdykoliv změnit datový typ proměnné.

Výhodou těchto jazyků je výrazně kratší kód (nejde jen o to, že není nutné psát deklarace proměnných) a vyšší přehlednost kódu, nevýhodou možnost výskytu běhových typových chyb při provozu programu.

K dynamicky typovaným jazykům řadíme například PYTHON, SMALLTALK včetně jeho volně šiřitelné implementace SQUEAK, LISP, RUBY a PROLOG.

Mezi staticky či dynamicky typovanými a silně či slabě typovanými jazyky není jednoznačný vztah, třebaže většina dynamicky typovaných jazyků provádí silné typové kontroly (za běhu programu). Například JAVA, C# a ADA jsou staticky silně typované, ale PYTHON dynamicky silně typovaný. Staticky typovaný jazyk se slabou typovou kontrolou je například C.

Příklad 4.12

Porovnáme kód funkce pro výpočet faktoriálu v několika jazycích.

JAVA – silně typový jazyk se statickou typovou kontrolou:

```
public class Factorial {
  public static long factorial (long n) {
    if (n == 0) return 1;
    else return n * factorial (n-1);
  }

  public static void main(String[] args) {
    long N = Long.parseLong(args[0]);
    System.out.println(faktorial(N));
  }
}
```

SCHEME – jazyk s dynamickou typovou kontrolou použitelný v umělé inteligenci, podobný LISPu:

```
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1)))))
```

TCL/TK – interpretovaný jazyk s dynamickou typovou kontrolou (vlastně vytváříme operátor `!`):

```
proc ! x {expr {$x<2 ? 1 : $x*[! [incr x - 1]]}}
```

PYTHON – interpretovaný jazyk s dynamickou typovou kontrolou:

```
def factorial(x):
    if x <= 0:
        return 1
    else:
        return x * factorial(x-1)
```

Úkoly ke kapitole 4

1. Je dán programovací jazyk s proměnnými těchto datových typů:

   - celá čísla,
   - racionální čísla,
   - řetězce,
   - pravdivostní hodnoty (boolean).

   Naprogramujte tabulku symbolů a její přístupové funkce pro tento jazyk (předpokládejme, že nemá blokovou strukturu). Jazyk má být interpretován, proto do tabulky ukládejte také hodnoty proměnných.

2. K jazyku z úkolu 1 přidejme názvy funkcí, které mají návratovou hodnotu, parametry se nepoužívají. Funkce mohou být vnořené, každá může mít vlastní lokální proměnné. Naprogramujte tabulku symbolů a její přístupové funkce.

3. Převeďte výraz $x = 3 - 2 + 1$ do všech tří základních druhů intermediálního kódu. Zkontrolujte výpočtem v intermediálním kódu, zda máte operátory ve správném pořadí.

4. Převeďte výraz $x = 25 - y * (8 + z)/2 + 5 * x$ do všech tří základních druhů intermediálního kódu.

5. Převeďte výraz IF $x < y + 2$ THEN $x := y + 2$ ELSE $x := z$ do všech tří základních druhů intermediálního kódu.

6. Jak víme, při konstrukci tabulky symbolů se často využívají metody známé z databází. Promyslete si, jak tímto způsobem (pomocí primárních klíčů) lze co nejvíce eliminovat neustálé porovnávání řetězců při procházení tabulky symbolů.

   Předpokládejme, že při zpracování deklarace proměnné (případně funkce nebo dalších možných prvků, které řadíme do tabulky symbolů) je této proměnné přiřazena nová hodnota primárního klíče reprezentovaná datovým typem, který se snadno porovnává (zřejmě číslem). V tabulce musíme během syntaktické analýzy evidovat také řetězcovou reprezentaci názvu proměnné, kterou používáme při zjištění proměnné mimo deklaraci, a nebo použijeme indexový soubor (seznam).

   Podle výše naznačeného řešení upravte kód tabulky symbolů pro interpretaci v příkladu 4.3 na straně 100. Přinese toto řešení zrychlení výpočtu i v případě, že v jazyce nejsou cykly a tedy nedochází k opakovanému vyhodnocování kódu dynamickou sémantikou?

KAPITOLA 5

Syntaxí řízený překlad

Až dosud jsme se věnovali spíše kontrole syntaktické a sémantické správnosti programu. V této kapitole se posuneme již dále směrem k samotnému překladu. Naučíme se řídit celý překlad syntaktickým analyzátorem, a to tak, aby na výstupu bylo také něco víc než jen posloupnost čísel pravidel, a také propojíme sémantiku se syntaxí pro interpretaci.

Zde nepřidáváme další fázi překladu, ale pouze propojujeme dříve probrané fáze, proto vstupy, výstupy a rozpoznávané chyby není třeba diskutovat.

5.1 Formální překlady a syntaxe

Následují základní definice související s překladem.

Definice 5.1 (Překlad) *Překlad z jazyka L_1 do jazyka L_2 je definován množinou uspořádaných dvojic [zdrojový_program, cílový_program], kde zdrojový_program $\in L_1^*$ a cílový_program $\in L_2^*$.*

Definice 5.2 (Formální překlad) *Formální překlad je binární relace $Z \subseteq D \times H$, která přiřazuje každému prvku z množiny D (zdrojový program) množinu prvků množiny H (jeho překladů).*

Pokud Z přiřadí pro každý prvek množiny D nejvýše jeden prvek množiny H, pak Z nazýváme funkcí *a překlad je* jednoznačný. *Zapisujeme $(x, y) \in Z$ nebo $Z(x) = y$ (pokud překlad není jednoznačný, pak píšeme $y \in Z(x)$).*

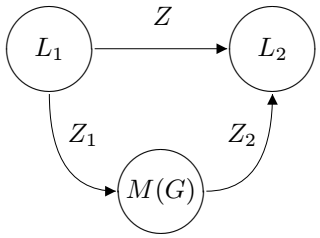
Definičním oborem *formálního překladu je množina všech hodnot, kterých může nabývat prvek x, tedy množina D,* oborem hodnot *je množina všech hodnot, kterých může nabývat prvek y, tedy množina H.*

Dále bude používáno toto značení:

- L_1 – vstupní jazyk, gramatika $G,\ L_1 = L(G)$,
- L_2 – výstupní jazyk,
- $M(G)$ – množina derivačních stromů všech slov generovaných gramatikou G.

Překlad určený kartézským součinem $Z \subseteq L_1 \times L_2$ je příliš složitý, proto ho rozdělíme na dva „podpřeklady" – Z_1 a Z_2.

První, $Z_1 \subseteq L_1 \times M(G)$, přeloží vstup z L_1 na derivační strom z $M(G)$, druhý, $Z_2 \subseteq M(G) \times L_2$, přeloží derivační strom z $M(G)$ na výstup z L_2.



Obrázek 5.1: Schéma formálního překladu

Totéž říká i následující definice:

Definice 5.3 (Syntaxí řízený překlad) *Syntaxí řízený překlad z jazyka L_1 do jazyka L_2 určený jako $Z \subseteq L_1 \times L_2$ je složení zobrazení $Z = Z_1 \circ Z_2$, kde Z_1 je překlad vstupního řetězce na ekvivalentní derivační strom a Z_2 je překlad tohoto derivačního stromu na řetězec výstupního jazyka.*

Syntaxí řízený překlad lze popsat překladovou gramatikou a realizovat překladovým automatem. Tento automat pak můžeme naprogramovat metodami, které vycházejí z metod ukázaných v předchozích kapitolách.

5.2 Překladová gramatika

Definice 5.4 (Překladová gramatika) *Překladová gramatika PG je uspořádaná pětice se zápisem $PG = (N, T, D, R, S)$, kde*

- *N je neprázdná konečná množina všech neterminálních symbolů gramatiky,*
- *T je neprázdná konečná množina vstupních terminálů – vstupní abeceda,*
- *D je konečná množina výstupních terminálů – výstupní abeceda (může být prázdná),*
- *R je neprázdná konečná množina pravidel ve tvaru $N \times (N \cup T \cup D)^*$,*
  jinak: $A \to \alpha,\ A \in N,\ \alpha \in (N \cup T \cup D)^*$,
- *S je startovací symbol gramatiky*

a platí $T \cap D = \emptyset$ (jsou disjunktní), a stejně jako v bezkontextové gramatice je $N \cap T = \emptyset$ a $N \cap D = \emptyset$.

Překladová gramatika generuje slova nad abecedou $(T \cup D)$, vstupní a výstupní terminály jsou „promíchané". My však potřebujeme odlišit vstup a výstup, k čemuž použijeme homomorfismus.

5.2.1 Vlastnosti překladových gramatik

Definice 5.5 (Homomorfismus) *Nechť Σ_1 a Σ_2 jsou abecedy. Homomorfismem nazýváme každé zobrazení $h : \Sigma_1^* \to \Sigma_2^*$ takové, že pro každé $a \in \Sigma_1^*,\ b \in \Sigma_1$ platí homomorfní podmínky:*

- $h(\varepsilon) = \varepsilon$,
- $h(a \cdot b) = h(a) \cdot h(b)$.

Homomorfismus se díky svým vlastnostem uplatňuje takto:

- rozložíme zdrojový řetězec na jednotlivé symboly,
- tyto symboly všechny přeložíme (uplatníme zobrazení h),
- výsledky překladu symbolů zřetězíme podle původního pořadí.

Homomorfní zobrazení použijeme ve formě vstupního a výstupního homomorfismu pro danou překladovou gramatiku.

Definice 5.6 (Vstupní a výstupní homomorfismus) *Nechť $PG = (N, T, D, R, S)$ je překladová gramatika. Vstupní a výstupní homomorfismus pro gramatiku PG jsou homomorfní zobrazení h_i (input – vstupní) a h_o (output – výstupní) s těmito vlastnostmi:*

$$h_i(X) = \begin{cases} X & ; \quad X \in (T \cup N) \\ \varepsilon & ; \quad X \in D \end{cases} \qquad h_o(X) = \begin{cases} X & ; \quad X \in (D \cup N) \\ \varepsilon & ; \quad X \in T \end{cases}$$

Z definice 5.6 je patrné, že úkolem vstupního homomorfismu je z daného řetězce odfiltrovat všechny výstupní symboly – řetězec se pak skládá pouze ze vstupních symbolů a případně neterminálů, a úkolem výstupního homomorfismu je naopak odfiltrovat všechny vstupní symboly.

Definice 5.7 (Překlad v překladové gramatice) *Překlad Z v překladové gramatice PG je definován jako množina uspořádaných dvojic vstupního a výstupního homomorfismu všech řetězců generovaných překladovou gramatikou:*

$$Z(PG) = \{(h_i(w), h_o(w))\ ;\ S \Rightarrow^* w,\ w \in (T \cup D)^*\}.$$

Překladová gramatika $PG = (N, T, D, R, S)$ se dá také chápat jako kombinace dvou gramatik, vstupní a výstupní.

Definice 5.8 (Vstupní a výstupní gramatika) *Nechť $PG = (N, T, D, R, S)$ je překladová gramatika.*

Vstupní gramatika *překladové gramatiky PG je gramatika $G_i = (N, T, P_i, S)$, kde platí* $P_i = \{A \to h_i(\alpha);\ (A \to \alpha) \in R\}$.

Výstupní gramatika *překladové gramatiky PG je gramatika $G_o = (N, D, P_o, S)$, kde platí* $P_o = \{A \to h_o(\alpha);\ (A \to \alpha) \in R\}$.

Poznámka: Vstupní i výstupní gramatika jsou běžné bezkontextové gramatiky, což značně zjednodušuje jejich další zpracování.

Jak víme, větná forma v bezkontextové gramatice je kterýkoliv člen derivací v této gramatice. Nyní definujeme podobné pojmy pro překladovou gramatiku.

Definice 5.9 (Formy v překladové gramatice) Překladová forma α *v překladové gramatice* $PG = (N, T, D, P, S)$ *je řetězec symbolů nad abecedou* $(N \cup T \cup D)$*, který lze v* PG *odvodit ze startovacího symbolu –* $S \Rightarrow^* \alpha$*. Překladová forma v překladové gramatice odpovídá větné formě v bezkontextové gramatice.*

Jestliže α *je překladová forma v překladové gramatice* PG*, pak* $h_i(\alpha)$ *je* vstupní větná forma *a* $h_o(\alpha)$ výstupní větná forma *v této gramatice.*

Poznámka: Vstupní větná forma překladové gramatiky je větnou formou vstupní gramatiky, výstupní větná forma překladové gramatiky je větnou formou výstupní gramatiky. Vstupní a výstupní větnou formu získáme z překladové formy uplatněním vstupního a výstupního homomorfismu.

V příkladu 5.1 najdeme překladovou gramatiku převádějící matematické výrazy z infixového tvaru na prefixový. Připomeneme si, jak vypadá výraz v infixu, prefixu a postfixu:

Infixový tvar	$(a + b) * c$	$a * (b + c)$
Prefixový tvar	$* + a\ b\ c$	$* \ a + b\ c$
Postfixový tvar	$a\ b + c *$	$a\ b\ c + *$

Tabulka 5.1: Vztah mezi infixem, prefixem a postfixem

Příklad 5.1

Vytvoříme překladovou gramatiku pro překlad infixových výrazů bez závorek na *prefixové*.

$PG = (\{A\}, \{n, +, *\},\ \{\textcircled{n}, \oplus, \circledast\}, R, A)$
s pravidly $A \rightarrow n + \oplus\textcircled{n}A \mid n * \circledast\textcircled{n}A \mid n\textcircled{n}$

V gramatice odvodíme slovo:

$A \Rightarrow n * \circledast\textcircled{n}A \Rightarrow n * \circledast\textcircled{n}n + \oplus\textcircled{n}A \Rightarrow n * \circledast\textcircled{n}n + \oplus\textcircled{n}n\textcircled{n}$

Na odvozené slovo použijeme vstupní a výstupní homomorfismus:

$h_i(n * \circledast\textcircled{n}n + \oplus\textcircled{n}n\textcircled{n}) \ = \ n * n + n$
$h_o(n * \circledast\textcircled{n}n + \oplus\textcircled{n}n\textcircled{n}) \ = \ \circledast\textcircled{n}\oplus\textcircled{n}\textcircled{n}$

Uspořádaná dvojice $(n * n + n,\ \circledast\textcircled{n}\oplus\textcircled{n}\textcircled{n})$ je prvkem překladu určeného překladovou gramatikou PG.

Pomocí vstupního a výstupního homomorfismu zjistíme také vstupní a výstupní gramatiku překladové gramatiky PG:

Vstupní gramatika:	*Výstupní gramatika:*
$G_i = (\{A\}, \{n, +, *\}, P_i, A)$	$G_o = (\{A\}, \{ⓝ, ⊕, ⊛\}, P_o, A)$
s pravidly $A \to n + A \mid n * A \mid n$	s pravidly $A \to ⊕ⓝA \mid ⊛ⓝA \mid ⓝ$

5.2.2 Speciální typy překladových gramatik

Protože podle gramatiky chceme vytvořit deterministicky pracující automat, budeme vyžadovat vlastnosti, které nám to ulehčí – překladová gramatika by měla být buď regulární nebo bezkontextová typu silná $LL(k)$ nebo silná $LR(k)$. Výhodnou vlastností je také zachovávání priorit operátorů.

Definice 5.10 (Regulární překladová gramatika) *Nechť $PG = (N, T, D, R, S)$ je překladová gramatika. PG je regulární, jestliže má pouze pravidla ve tvaru*

- $A \to x\gamma B,\ A, B \in N,\ x \in T,\ \gamma \in D^*$,
- $A \to x\gamma,\ A \in N,\ x \in T,\ \gamma \in D^*$,
- $S \to \varepsilon$, *jestliže S se nevyskytuje na pravé straně žádného pravidla.*

Z definice 5.10 je zřejmé, že *vstupní gramatikou* regulární překladové gramatiky je regulární gramatika Chomského hierarchie a *výstupní gramatikou* je rozšířená regulární gramatika (připouštějí se také pravidla s pravou stranou nad množinami D^*N a D^*).

Definice 5.11 (Překladová gramatika typu silná $LL(k)$ / $LR(k)$) *Překladová gramatika PG je typu silná $LL(k)$ (silná $LR(k)$) pro nějaké přirozené číslo k, jestliže její vstupní gramatika je typu silná $LL(k)$ (resp. silná $LR(k)$).*

Příklad 5.2

Vytvoříme překladovou gramatiku pro překlad infixových výrazů na *postfixové*, která zachovává priority operátorů.

$PG = (\{E, T, F\},\ \{n, +, *, (,)\},\ \{ⓝ, ⊕, ⊛\},\ R, E)$

$E \to E + T⊕ \mid T$

$T \to T * F⊛ \mid F$

$F \to nⓝ \mid (E)$

Ukázka odvození slova:

$E \Rightarrow E + T⊕ \Rightarrow T + T⊕ \Rightarrow T * F⊛ + T⊕ \Rightarrow F * F⊛ + T⊕ \Rightarrow nⓝ * F⊛ + T⊕ \Rightarrow$

$\Rightarrow nⓝ * nⓝ⊛ + T⊕ \Rightarrow nⓝ * nⓝ⊛ + F⊕ \Rightarrow nⓝ * nⓝ⊛ + nⓝ⊕$

Použijeme vstupní a výstupní homomorfismus:

$h_i(n\textcircled{n} * n\textcircled{n}\textcircled{*} + n\textcircled{n}\textcircled{+}) = n * n + n$

$h_o(n\textcircled{n} * n\textcircled{n}\textcircled{*} + n\textcircled{n}\textcircled{+}) = \textcircled{n}\textcircled{n}\textcircled{*}\textcircled{n}\textcircled{+}$

Vstupní gramatika:	*Výstupní gramatika:*
$G_i = (\{E, T, F\},\ \{n, +, *, (,)\},\ P_i, E)$	$G_o = (\{E, T, F\},\ \{\textcircled{n}, \textcircled{+}, \textcircled{*}\},\ P_o, E)$
$E \to E + T \mid T$	$E \to ET\textcircled{+} \mid T$
$T \to T * F \mid F$	$T \to TF\textcircled{*} \mid F$
$F \to n \mid (E)$	$F \to \textcircled{n} \mid E$

Vstupní gramatika sice zachovává priority operátorů, ale není $LL(1)$, proto ani překladová gramatika PG není $LL(1)$.

Příklad 5.3

Překladovou gramatiku pro překlad infixových výrazů na postfixové z předchozího příkladu upravíme tak, aby byla typu $LL(1)$.

$PG = (\{E, A, T, B, F\},\ \{n, +, *, (,)\},\ \{\textcircled{n}, \textcircled{+}, \textcircled{*}\},\ R, E)$

$E \to TA$

$A \to +T\textcircled{+}A \mid \varepsilon$

$T \to FB$

$B \to *F\textcircled{*}B \mid \varepsilon$

$F \to n\textcircled{n} \mid (E)$

Vstupní gramatika:

$G_i = (\{E, A, T, B, F\},\ \{n, +, *, (,)\},\ P_i, E)$

$E \to TA$	$\text{FOLLOW}(E) = \{\$,)\}$
$A \to +TA \mid \varepsilon$	$\text{FOLLOW}(A) = \{\$,)\}$
$T \to FB$	$\text{FOLLOW}(T) = \{+, \$,)\}$
$B \to *FB \mid \varepsilon$	$\text{FOLLOW}(B) = \{+, \$,)\}$
$F \to n \mid (E)$	$\text{FOLLOW}(F) = \{*, +, \$,)\}$

Podle vzorců probíraných v předchozích kapitolách lze dokázat, že vstupní gramatika je $LL(1)$, proto také překladová gramatika PG je $LL(1)$. V gramatice je obsaženo vše potřebné pro běžné matematické výrazy včetně závorek a priority operátorů, lze také snadno přidat operátory $-$ a $/$.

Příklad 5.4

Vytvoříme *regulární* překladovou gramatiku pro překlad infixových výrazů na *postfixové* bez závorek.

$PG = (\{S, A, B, C\},\ \{n, +, *\},\ \{ⓝ, ⊕, ⊛\},\ R, S)$
$S \to nⓝA \mid nⓝ$
$A \to *B \mid +C$
$B \to nⓝ⊛A \mid nⓝ⊛$
$C \to nⓝ⊕A \mid nⓝ⊕$

Ukázka odvození:

$S \Rightarrow nⓝA \Rightarrow nⓝ * B \Rightarrow nⓝ * nⓝ⊛$

Vstupní gramatika:

$PG = (\{S, A, B, C\},\ \{n, +, *\},\ P_i, S)$
$S \to nA \mid n$
$A \to *B \mid +C$
$B \to nA \mid n$
$C \to nA \mid n$

Výstupní gramatika:

$G_o = (\{S, A, B, C\},\ \{ⓝ, ⊕, ⊛\},\ P_o, S)$
$S \to ⓝA \mid ⓝ$
$A \to B \mid C$
$B \to ⓝ⊛A \mid ⓝ⊛$
$C \to ⓝ⊕A \mid ⓝ⊕$

Vstupní gramatika je regulární, proto i PG je regulární. Se závorkami si regulární gramatiky neporadí (přesněji – závorky by mohly být, ale nikoliv vnořené). Nevýhodou je také chybějící řešení priority operátorů, proto tato gramatika není vhodná pro vyhodnocování, pouze pro přeložení výrazu do postfixu.

5.3 Překladový automat

Překladový automat budeme konstruovat pro zadaný překlad $Z \subseteq L_1 \times L_2$. Účelem je pro každý vstup $h_i(w) \in L_1$ vytvořit na výstupu $h_o(w) \in L_2$. Postup bude podobný jako u překladového automatu pro běžnou syntaktickou analýzu, jen budeme více využívat výstupní pásku. Rozlišujeme

- *konečný překladový automat* pro regulární překladovou gramatiku,
- *zásobníkový překladový automat* pro bezkontextovou překladovou gramatiku.

5.3.1 Konečný překladový automat

Definice 5.12 (Konečný překladový automat) *Konečný překladový automat je uspořádaná šestice $KPA = (Q, T, D, \delta, q_0, F)$, kde*

- *Q je konečná neprázdná množina stavů automatu,*
- *T je konečná neprázdná množina vstupních symbolů,*
- *D je konečná množina výstupních symbolů, $D \cap T = \emptyset$,*

- *δ je přechodová funkce automatu, obecně nedeterministická, $\delta : Q \times T \to \mathcal{P}(Q \times D^*)$ (u nedeterministického automatu je výsledkem zobrazení množina uspořádaných dvojic stavů a výstupních řetězců),*
- *$q_0 \in Q$ je počáteční stav,*
- *$F \subseteq Q$ je množina koncových stavů automatu.*

Konfigurace *konečného překladového automatu $KPA = (Q, T, D, \delta, q_0, F)$ je uspořádaná trojice $(q, \alpha, \beta) \in Q \times T^* \times D^*$.* Počáteční konfigurace *je (q_0, w, ε), kde w je vstupní řetězec (řetězec vstupních symbolů),* koncová konfigurace *je (q_f, ε, y), kde $q_f \in F$, y je výstupní řetězec (řetězec výstupních symbolů).*

Činnost automatu $KPA = (Q, T, D, \delta, q_0, F)$ probíhá takto:

- na vstupu je vstupní řetězec složený ze symbolů množiny T, automat postupně čte vstupní řetězec a podle znaků v tomto řetězci přechází mezi stavy,
- při každém přechodu může na výstupní pásku přidat řetězec výstupních symbolů z množiny D,
- výpočet končí tehdy, když je přečteno vstupní slovo a automat je v některém z koncových stavů.

Podle popisu činnosti automatu je *relace přechodu mezi konfiguracemi* určena následovně: dvě libovolné konfigurace konečného překladového automatu $(p, a\xi, \beta)$ a $(q, \xi, \beta\gamma)$ jsou v relaci $\vdash$, pokud $\delta(p, a) \ni (q, \gamma)$, píšeme $(p, a\xi, \beta) \vdash (q, \xi, \beta\gamma)$.

Definice 5.13 (Překlad konečného překladového automatu) *Překlad definovaný konečným překladovým automatem $KPA = (Q, T, D, \delta, q_0, F)$ je množina uspořádaných dvojic*

$$Z(KPA) = \{(u, v) \mid (q_0, u, \varepsilon) \vdash^* (q_f, \varepsilon, v),\ q_f \in F\}.$$

Příklad 5.5

Sestrojíme konečný překladový automat realizující inverzní zobrazení nad abecedou $\{0, 1\}$.
$KPA = (\{q\},\ \{0, 1\},\ \{⓪, ①\},\ \delta,\ q,\ \{q\})$, s přechodovou funkcí δ:

$\delta(q, 0) = (q, ①)$
$\delta(q, 1) = (q, ⓪)$

	0	1
q	①	⓪

Kdyby v automatu existovalo více stavů, musely by být uvedeny v buňkách tabulky.

Ukázka zpracování slova 110101:

$(q, 110101, \varepsilon) \vdash (q, 10101, ⓪) \vdash (q, 0101, ⓪⓪) \vdash (q, 101, ⓪⓪①) \vdash (q, 01, ⓪⓪①⓪) \vdash$
$\vdash (q, 1, ⓪⓪①⓪①) \vdash (q, \varepsilon, ⓪⓪①⓪①⓪)$

U některých překladů dokážeme sestrojit konečný překladový automat přímo, jak jsme viděli na příkladu 5.5. Obvykle je však jednodušší popsat strukturu vstupního a výstupního jazyka překladovou gramatikou a podle ní pak vytvořit překladový automat.

Podle překladové gramatiky $PG = (N, T, D, R, S)$ sestrojíme konečný překladový automat $KPA = (Q, T, D, \delta, q_0, F)$ následovně:

- $Q = N \cup \{X\}$, $X \notin N$ je nově přidaný stav,
- množiny T a D jen přejmeme jako vstupní a výstupní abecedu,
- $q_0 = S$,
- jestliže $A \to a\gamma B,\ a \in T,\ \gamma \in D^*$ je pravidlo gramatiky PG, tak definujeme $\delta(A, a) \ni (B, \gamma)$
- jestliže $A \to a\gamma,\ a \in T,\ \gamma \in D^*$ je pravidlo gramatiky PG, tak definujeme $\delta(A, a) \ni (X, \gamma)$
- jestliže $\varepsilon \in L(PG)$, pak $F = \{S, X\}$, jinak $F = \{X\}$.

Nově přidaný stav X nám slouží jako koncový stav, do tohoto stavu vedeme výpočetní cesty, které byly v gramatice ukončeny použitím terminálního pravidla (takového, které na pravé straně neobsahuje neterminál).

Příklad 5.6

Podle regulární překladové gramatiky sestrojíme konečný překladový automat.

$PG = (\{S, A, B, C\},\ \{n, +, *\},\ \{\textcircled{n}, \oplus, \circledast\},\ R, S)$

$S \to n\textcircled{n}A \mid n\textcircled{n}$
$A \to *B \mid +C$
$B \to n\textcircled{n}\circledast A \mid n\textcircled{n}\circledast$
$C \to n\textcircled{n}\oplus A \mid n\textcircled{n}\oplus$

$KPA = (\{S, A, B, C, X\},\ \{n, +, *\},\ \{\textcircled{n}, \oplus, \circledast\},\ \delta,\ S,\ \{X\})$,

$\delta(S, n) = \{(A, \textcircled{n}), (X, \textcircled{n})\}$
$\delta(A, *) = \{(B, \varepsilon)\}$
$\delta(A, +) = \{(C, \varepsilon)\}$
$\delta(B, n) = \{(A, \textcircled{n}\circledast), (X, \textcircled{n}\circledast)\}$
$\delta(C, n) = \{(A, \textcircled{n}\oplus), (X, \textcircled{n}\oplus)\}$

	a	$+$	$*$
S	$(A, \textcircled{n}), (X, \textcircled{n})$		
A		(C, ε)	(B, ε)
B	$(A, \textcircled{n}\circledast), (X, \textcircled{n}\circledast)$		
C	$(A, \textcircled{n}\oplus), (X, \textcircled{n}\oplus)$		

5.3.2 Zásobníkový překladový automat

Definice 5.14 (Zásobníkový překladový automat) *Zásobníkový překladový automat je uspořádaná 8-ce $ZPA = (Q, T, \Gamma, D, \delta, q_0, Z_0, F)$, kde*

- *Q je neprázdná konečná množina vnitřních stavů,*
- *T je neprázdná konečná množina vstupních symbolů,*

- Γ *je konečná množina zásobníkových symbolů,*
- D *je konečná množina výstupních symbolů (může být prázdná),* $D \cap T = \emptyset$*,*
- δ *je zobrazení* $\delta : \ Q \times (T \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^* \times D^*)$ *(přechodová funkce automatu),*
- $q_0 \in Q$ *je počáteční stav,*
- $Z_0 \in \Gamma$ *je počáteční symbol na zásobníku,*
- $F \subseteq Q$ *je množina koncových stavů.*

Konfigurace *zásobníkového překladového automatu má tvar* $(q, \alpha, \gamma, \beta) \in Q \times T^* \times \Gamma^* \times D^*$. Počáteční konfigurace *je* $(q_0, w, Z_0, \varepsilon)$*, kde* $w \in T^*$ *je vstupní řetězec.*

Relace přechodu mezi konfiguracemi *je určena následovně: dvě libovolné konfigurace zásobníkového překladového automatu* $(p, a\xi, A\gamma, \beta)$ *a* $(q, \xi, \vartheta\gamma, \beta\omega)$ *nad* $Q \times T^* \times \Gamma^* \times D^*$ *jsou v relaci* $\vdash$*, pokud* $\delta(p, a, A) \ni (q, \vartheta, \omega)$*, píšeme* $(p, a\xi, A\gamma, \beta) \vdash (q, \xi, \vartheta\gamma, \beta\omega)$.

Koncová konfigurace *je definována podobně jako u běžného zásobníkového automatu podle jeho typu – pro zásobníkový automat končící s prázdným zásobníkem je* $(q, \varepsilon, \varepsilon, \beta), \ F = \emptyset$*, a pro zásobníkový automat končící v koncovém stavu je* $(q, \varepsilon, \gamma, \beta)$, $q \in Q, \ \beta \in D^*, \ \gamma \in \Gamma^*$.

Definice 5.15 (Překlad zásobníkového překladového automatu) *Překlad definovaný zásobníkovým překladovým automatem* $ZPA = (Q, T, \Gamma, D, \delta, q_0, Z_0, F)$ *je množina uspořádaných dvojic*

$$Z(ZPA) = \{(u, v) \mid (q_0, u, Z_0, \varepsilon) \ \vdash^* \ (q, \varepsilon, \alpha, v), \text{ kde } (q, \varepsilon, \alpha, v) \text{ je koncová konfigurace}\}.$$

Máme překladovou gramatiku $PG = (N, T, D, R, S)$. Sestrojíme překladový zásobníkový automat $ZPA = (Q, T, \Gamma, D, \delta, q_0, Z_0, \emptyset)$.

- $Q = \{q\}$, $\Gamma = N \cup T \cup D$,
- $q_0 = q$, $Z_0 = S$,
- δ funkce:
  - Podle pravidel: $A \to \alpha$ v R — $\delta(q, \varepsilon, A) \ni (q, \alpha, \varepsilon)$
  - Vstupy: pro každé $a \in T$ — $\delta(q, a, a) = (q, \varepsilon, \varepsilon)$
  - Výstupy: pro každé $\textcircled{a} \in D$ — $\delta(q, \varepsilon, \textcircled{a}) = (q, \varepsilon, \textcircled{a})$

Příklad 5.7

Pro zadanou překladovou gramatiku vytvoříme překladový automat.

$PG = (\{S, A, B, C, D\},$
$\{n, +, -, *, /, (,)\},$
$\{\textcircled{n}, \oplus, \ominus, \circledast, \textcircled{/}\},$
$R, S)$

Pravidla v R:

$S \to AB$
$A \to CD$
$B \to +A\oplus B \mid -A\ominus B \mid \varepsilon$
$C \to (S) \mid i\textcircled{i} \mid n\textcircled{n}$
$D \to *C\circledast D \mid /C\textcircled{/}D \mid \varepsilon$

$ZPA = (\{q\},\ \{n, +, -, *, /, (,)\},\ \Gamma,\ \{ⓝ, ⊕, ⊖, ⊛, ⊘\}, \delta, q, S, \emptyset)$, kde
$\Gamma = \{S, A, B, C, D, n, +, -, *, /, (,), ⓝ, ⊕, ⊖, ⊛, ⊘\}$, δ-funkce je

$\delta(q, S, \varepsilon) = \{(q, AB, \varepsilon)\}$
$\delta(q, A, \varepsilon) = \{(q, CD, \varepsilon)\}$
$\delta(q, B, \varepsilon) = \{(q, +A⊕B, \varepsilon), (q, -A⊖B, \varepsilon), (q, \varepsilon, \varepsilon)\}$
$\delta(q, C, \varepsilon) = \{(q, (S), \varepsilon), (q, iⓘ, \varepsilon), (q, nⓝ, \varepsilon)\}$
$\delta(q, D, \varepsilon) = \{(q, *C⊛D, \varepsilon), (q, /C⊘D, \varepsilon), (q, \varepsilon, \varepsilon)\}$

$\delta(q, n, n) = \{(q, \varepsilon, \varepsilon)\}$ $\quad$ $\delta(q, \varepsilon, ⓝ) = \{(q, \varepsilon, ⓝ)\}$
$\delta(q, +, +) = \{(q, \varepsilon, \varepsilon)\}$ $\quad$ $\delta(q, \varepsilon, ⊕) = \{(q, \varepsilon, ⊕)\}$
$\delta(q, -, -) = \{(q, \varepsilon, \varepsilon)\}$ $\quad$ $\delta(q, \varepsilon, ⊖) = \{(q, \varepsilon, ⊖)\}$
$\delta(q, *, *) = \{(q, \varepsilon, \varepsilon)\}$ $\quad$ $\delta(q, \varepsilon, ⊛) = \{(q, \varepsilon, ⊛)\}$
$\delta(q, /, /) = \{(q, \varepsilon, \varepsilon)\}$ $\quad$ $\delta(q, \varepsilon, ⊘) = \{(q, \varepsilon, ⊘)\}$
$\delta(q, (, () = \{(q, \varepsilon, \varepsilon)\}$
$\delta(q,),)) = \{(q, \varepsilon, \varepsilon)\}$

V příkladu 5.7 jsme vytvořili překladový automat, který není deterministický. Pro $LL(1)$ překladové gramatiky však můžeme sestrojit rozkladovou tabulku:

- vypočteme potřebné množiny FIRST a FOLLOW *vstupní gramatiky*,
- ověříme, zda je vstupní gramatika typu $LL(1)$,
- vytvoříme rozkladovou tabulku vstupní gramatiky (výstupní symboly se nemohou objevit na vstupní pásce, podle které se řídíme).

Rozkladovou tabulku tedy vytváříme stejným způsobem, jako u běžné bezkontextové gramatiky. Rozdíl je jen v tom, že v zásobníku se mohou nacházet také výstupní symboly. Jestliže ze zásobníku vyjmeme výstupní symbol, pak tento symbol pouze přepíšeme na výstupní pásku.

Příklad 5.8

Pro gramatiku z předchozího příkladu vytvoříme rozkladovou tabulku.

$PG = (\{S, A, B, C, D\},\ \{n, +, -, *, /, (,)\},\ \{ⓝ, ⊕, ⊖, ⊛, ⊘\},\ R, S)$

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

Rozkladovou tabulku vytvoříme podle vstupní gramatiky, najdeme ji na následující stránce.

	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

Tabulku použijeme při zpracování slova $n + i * n$ (stav q nebudeme psát, nemění se). Začneme v počáteční konfiguraci a řídíme se vždy symbolem na vstupu (určuje sloupec tabulky) a symbolem na vrcholu zásobníku (určuje její řádek). V buňce zjištěného sloupce a řádku je akce, kterou automat provede, obvykle expanze podle pravidla s daným číslem.

$(n + i * n\$, S\#, \varepsilon) \vdash (n + i * n\$, AB\#, \varepsilon) \vdash (n + i * n\$, CDB\#, \varepsilon) \vdash$
$\vdash (n + i * n\$, n\textcircled{n}DB\#, \varepsilon) \vdash (+i * n\$, \textcircled{n}DB\#, \varepsilon) \vdash (+i * n\$, DB\#, \textcircled{n}) \vdash$
$\vdash (+i * n\$, B\#, \textcircled{n}) \vdash (+i * n\$, +A\oplus B\#, \textcircled{n}) \vdash (i * n\$, A\oplus B\#, \textcircled{n}) \vdash$
$\vdash (i * n\$, CD\oplus B\#, \textcircled{n}) \vdash (i * n\$, i\textcircled{i}D\oplus B\#, \textcircled{n}) \vdash (*n\$, \textcircled{i}D\oplus B\#, \textcircled{n}) \vdash$
$\vdash (*n\$, D\oplus B\#, \textcircled{n}\textcircled{i}) \vdash (*n\$, *C\circledast D\oplus B\#, \textcircled{n}\textcircled{i}) \vdash (n\$, C\circledast D\oplus B\#, \textcircled{n}\textcircled{i}) \vdash$
$\vdash (n\$, n\textcircled{n}\circledast D\oplus B\#, \textcircled{n}\textcircled{i}) \vdash (\$, \textcircled{n}\circledast D\oplus B\#, \textcircled{n}\textcircled{i}) \vdash (\$, \circledast D\oplus B\#, \textcircled{n}\textcircled{i}\textcircled{n}) \vdash$
$\vdash (\$, D\oplus B\#, \textcircled{n}\textcircled{i}\textcircled{n}\circledast) \vdash (\$, \oplus B\#, \textcircled{n}\textcircled{i}\textcircled{n}\circledast) \vdash (\$, B\#, \textcircled{n}\textcircled{i}\textcircled{n}\circledast\oplus) \vdash (\$, \#, \textcircled{n}\textcircled{i}\textcircled{n}\circledast\oplus)$

5.4 Atributová překladová gramatika

Nadále budeme počítat s tím, že překlad bude řízen syntaktickým analyzátorem, navíc přidáme zpracování sémantiky.

5.4.1 Atributy a sémantická pravidla

Symbolům přiřadíme *atributy*, které budou představovat jejich sémantické vlastnosti. Symboly si můžeme představit jako záznamy (struktury) a jejich atributy jako prvky těchto záznamů – podobně k nim také budeme přistupovat (připomeňme datový typ `TSymbol`).

Tak jako jsme k symbolům přidali jejich atributy, syntaktická pravidla obohatíme o *sémantická pravidla*, ve kterých budou atributy symbolů zpracovávány. Každé sémantické pravidlo bude patřit k určitému konkrétnímu syntaktickému pravidlu (jedno syntaktické pravidlo může mít jakýkoliv počet sémantických pravidel), dokonce někdy bude nutné sémantické pravidlo umístit na přesně danou pozici v syntaktickém pravidle.

Překladovou gramatiku obohacenou o atributy a sémantická pravidla nazýváme atributovou překladovou gramatikou. Zatímco překladové gramatiky bylo možné naprogramovat pro kompletní syntaktickou analýzu včetně generování výstupu, atributové gramatiky použijeme pro naprogramování jak syntaxe, tak i sémantiky programovacího jazyka.

Definice 5.16 (Atributová překladová gramatika) *Atributová překladová gramatika je trojice* $APG = (PG, A, F)$*, kde*

- $PG = (N, T, D, R, S)$ *je překladová gramatika,*
- A *je množina atributů přiřazených symbolům z množiny* $N \cup T \cup D$*,*
- F *je množina sémantických pravidel, z nichž každé přísluší k určenému syntaktickému pravidlu překladové gramatiky.*

Jestliže je množina výstupních symbolů D *prázdná, hovoříme pouze o atributové gramatice.*

Fakt, že symbol A má přiřazeny atributy val, m a p, zapisujeme $A[val, m, p]$. Pokud je za symbolem prázdná závorka, znamená to, že tento symbol nemá žádné atributy. K atributům přistupujeme stejně jako k záznamům (strukturám) v programovacích jazycích, například $A.val = 2$.

Řetězec symbolů, kterým jsou přiřazeny atributy, budeme nazývat *atributovaný řetězec*.

Definice 5.17 (Vstupní a výstupní atributovaný řetězec) *Nechť* w *je slovo generované atributovou překladovou gramatikou* $APG = (PG, A, F)$*. Potom* $h_i(w)$*, kde* h_i *je vstupní homomorfismus v překladové gramatice* PG*, je řetězec skládající se ze vstupních terminálních symbolů ohodnocených atributy a nazýváme ho* vstupní atributovaný řetězec.

Obdobně $h_o(w)$*, kde* h_o *je výstupní homomorfismus v* PG*, je řetězec skládající se z výstupních terminálních symbolů opět ohodnocených atributy a nazýváme ho* výstupní atributovaný řetězec.

Definice 5.18 (Atributový překlad) *Atributovaný překlad v atributové překladové gramatice je množina uspořádaných dvojic vstupních a výstupních atributovaných řetězců.*

Atributové gramatiky se používají také k interpretaci výrazů, jak uvidíme v následujícím příkladu.

Příklad 5.9

Bezkontextovou gramatiku rozšíříme o atributy a sémantická pravidla tak, aby popisovala výpočet matematických výrazů s použitím priority operátorů.

$S \to i = A$
$A \to A + B \mid A - B \mid B$
$B \to B * C \mid B/C \mid C$
$C \to (A) \mid n \mid i$

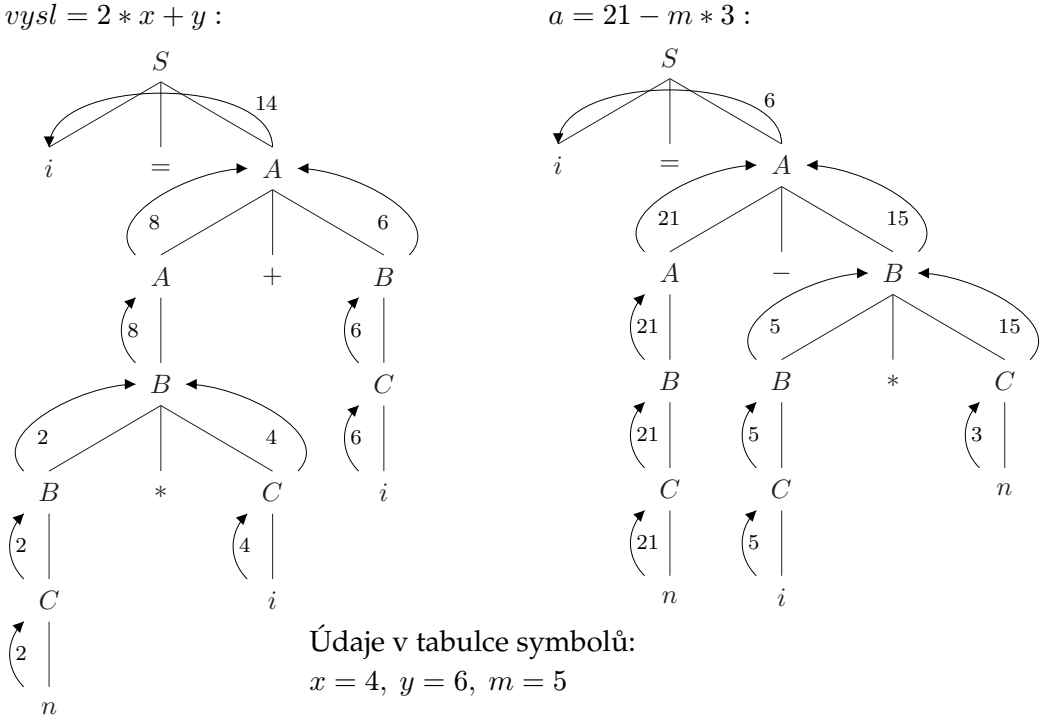
Použijeme atributy $S[\,]$, $A[val]$, $B[val]$, $C[val]$, $n[lex]$, $i[nazev]$, $+[\,]$, $*[\,]$, ...

Atributy pojmenované val obsahují výsledek či mezivýsledek, $n.lex$ vrací hodnotu čísla načtenou lexikálním analyzátorem (atribut), symbol i má atribut $nazev$ (název proměnné) získaný z tabulky symbolů nebo od lexikálního analyzátoru, operátory jsou bez atributů.

Nyní vytvoříme potřebná sémantická pravidla a přidáme je k syntaktickým pravidlům gramatiky. Protože se v některých pravidlech vyskytuje tentýž symbol vícekrát, je třeba

výskyty tohoto symbolu očíslovat dolními indexy, abychom dokázali rozlišit, kterého symbolu se týká konkrétní sémantická funkce.

Syntaktická pravidla:	*K nim příslušná sémantická pravidla:*
$S \to i = A$	$\text{Uloz}(i.nazev, A.val)$
$A_0 \to A_1 + B$	$A_0.val = A_1.val + B.val$
$A_0 \to A_1 - B$	$A_0.val = A_1.val - B.val$
$A \to B$	$A.val = B.val$
$B_0 \to B_1 * C$	$B_0.val = B_1.val * C.val$
$B_0 \to B_1/C$	$B_0.val = \text{if}(C.val <> 0)\text{then } B_1.val/C.val \text{ else error}$
$B \to C$	$B.val = C.val$
$C \to (A)$	$C.val = A.val$
$C \to n$	$C.val = n.lex$
$C \to i$	$C.val = \text{ZjistiHodnotu}(i.nazev)$



Obrázek 5.2: Derivační stromy se zachycením toku hodnot v atributech

Sémantická pravidla nám říkají, co se má dít ve chvíli, kdy se vyhodnocuje určité syntaktické pravidlo. Například při vyhodnocování pravidla $A \to A + B$ se po (rekurzivním) výpočtu sečtou hodnoty získané z podstromů na pravé straně pravidla a uloží se do atributu symbolu na levé straně pravidla (tj. symbolu ve vyšším patře derivačního stromu, hodnota se posílá po stromě nahoru).

Na obrázku 5.2 je ukázán tok hodnot v atributech symbolů v derivačních stromech dvou různých výrazů. Jak vidíme, hodnoty se posílají nahoru ke kořeni stromu, kde jsou přeposlány do proměnné na levé straně přiřazovacího příkazu. V uzlech, které mají více než jednoho potomka, se provádějí průběžné výpočty.

Sémantická pravidla jsou nejen algebraická, ale mohou mít podobu jakékoliv funkce, kterou dokážeme naprogramovat. Například funkce Uloz a ZjistiHodnotu slouží k přístupu do tabulky symbolů, funkce error hlásí (vypisuje) chybu.

Příklad 5.10

Sestavíme atributovou gramatiku pro překlad infixového výrazu na postfixový. Překladovou gramatiku převezmeme z příkladu 5.8 na straně 127.

$PG = (\{S, A, B, C, D\},\ \{n, +, -, *, /, (,)\},\ \{\textcircled{n}, \oplus, \ominus, \circledast, \textcircled{/}\},\ R, S)$

$S \to AB$

$A \to CD$

$B \to +A\oplus B \mid -A\ominus B \mid \varepsilon$

$C \to (S) \mid i\textcircled{i} \mid n\textcircled{n}$

$D \to *C\circledast D \mid /C\textcircled{/} D \mid \varepsilon$

Vstupní symboly pro operátory a závorky budeme jen načítat, jejich zpracování se v sémantice téměř nijak neprojeví, pouze ze symbolů pro názvy proměnných a čísla si vezmeme atributy, abychom je mohli zapsat do výstupního řetězce.

Pro výstup použijeme sémantickou funkci, kterou pojmenujeme output(). Tato funkce jednoduše vypíše svůj parametr (typu řetězec) na výstup (třeba do výstupního souboru nebo do editačního okna).

Pokud bychom však chtěli s výstupem gramatiky (resp. automatu nebo programu) dále pracovat, bylo by vhodnější této funkci předávat symboly, které by byly zařazovány například do dynamické struktury, ze které je lze získat bez dalších konverzí mnohem snadněji než z textového řetězce.

$S \to AB$

$B \to +A\oplus\ \{\text{output}('+')\}\ B$

$B \to A\ominus\ \{\text{output}('-')\}\ B$

$B \to \varepsilon$

$A \to CD$

$C \to (S)$

$C \to i\textcircled{i}\ \{\text{output}(i.nazev)\}$

$C \to n\textcircled{n}\ \{\text{output}(n.val)\}$

$D \to *C\circledast\ \{\text{output}('*')\}\ D$

$D \to /C\textcircled{/}\ \{\text{output}('/')\}\ D$

$D \to \varepsilon$

Sémantické funkce pro výstup musí být provedeny vždy hned po vyhodnocení příslušného výstupního terminálu, protože v podstromě následujícího neterminálu může dojít k dalším voláním podobných funkcí. Proto sémantická pravidla umístíme dovnitř syntaktických a pro lepší odlišení je uzavřeme do lomených závorek.

5.4.2 Typy atributů

Z obrázku 5.2 je zřejmé, že přes atributy se v derivačním stromě posílají údaje. V atributové gramatice z příkladu 5.9 jsou tyto údaje posílány výhradně směrem nahoru, ale existují atributové překladové gramatiky, ve kterých jsou data posílána směrem dolů. Směr toku dat je důležitý především pro implementaci, proto budeme rozlišovat dva typy atributů a každý z nich pak jinak naprogramujeme.

Definice 5.19 (Syntetizované a dědičné atributy) Syntetizovaný atribut *je atribut, jehož hodnota závisí na hodnotách atributů uzlů jeho vlastního podstromu v derivačním stromě (tedy jeho následníků a rekurzívně směrem dolů). Syntetizované atributy slouží k posílání údajů v derivačním stromě směrem zdola nahoru.*

Dědičný atribut *je atribut, jehož hodnota závisí na hodnotách atributů nadřízeného uzlu nebo uzlů na stejné úrovni derivačního stromu vlevo od tohoto symbolu.*

Žádný atribut nemůže být zároveň syntetizovaný a dědičný, množiny syntetizovaných a dědičných atributů jsou navzájem disjunktní.

Například v pravidle $A \to B_1 \ldots B_i \ldots B_n$:

- syntetizované atributy symbolu A mohou být vypočteny ze syntetizovaných atributů symbolů $B_1, \ldots, B_n$,
- syntetizované atributy symbolů B_i nemohou záviset na atributech ostatních symbolů pravidla (jsou vypočteny v pravidle, které přepisuje symbol B_i – přesněji v podstromu symbolu B_i),
- dědičné atributy symbolu A nejsou závislé na atributech symbolů B_i,
- dědičné atributy některého symbolu B_i mohou záviset na dědičných atributech symbolu A a (jakýchkoliv) atributech symbolů $B_1, \ldots, B_{i-1}$,
- dědičné atributy symbolu B_i nemohou záviset na (žádných) atributech symbolů $B_{i+1}, \ldots, B_n$, protože tyto symboly jsou vyhodnocovány až po B_i a hodnota jejich atributů není při vyhodnocování B_i známa.

Oba typy atributů musí být někde inicializovány. Hodnoty syntetizovaných atributů postupují v derivačním stromě zdola nahoru, proto jejich inicializace probíhá většinou v listech derivačního stromu, tedy v terminálních pravidlech, a obvykle souvisí s činností lexikálního analyzátoru.

U dědičných atributů si musíme dát na inicializaci mnohem větší pozor. Většinou (ne vždy) je třeba dědičné atributy inicializovat v kořeni derivačního stromu, tedy při použití prvního pravidla v derivaci.

Syntetizované atributy bývají inicializovány lexikálním analyzátorem, ale ne vždy. Případ, kdy při návrhu gramatiky je třeba zajistit inicializaci syntetizovaného atributu, vidíme v příkladu 5.11.

Příklad 5.11

Vytvoříme atributovou překladovou gramatiku takovou, že v atributu výstupního terminálu bude počet prvků seznamu.

Nejdřív sestavíme překladovou gramatiku generující seznam čísel s jedním výstupním terminálem v.

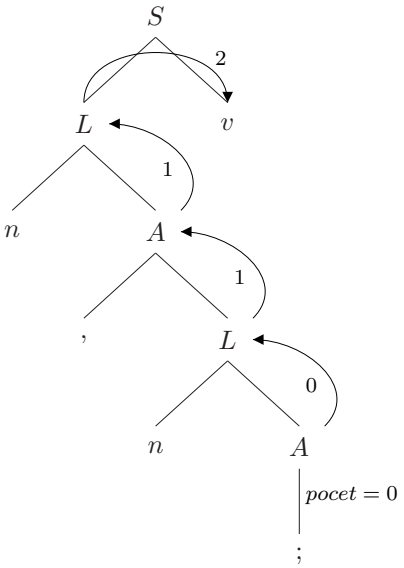
$S \to Lv$
$L \to nA$
$A \to , L \mid ;$

Použijeme atributy $v[vysl],\ L[pocet],\ A[pocet]$.

V syntetizovaném atributu $pocet$ budeme směrem nahoru posílat počet prvků seznamu v daném podstromě, při použití pravidla $L \to nA$ se počet zvýší o 1. Atribut inicializujeme v terminálním pravidle $A \to ;$.

$S \to Lv$	$v.vysl = L.pocet$
$L \to nA$	$L.pocet = A.pocet + 1$
$A \to , L$	$A.pocet = L.pocet$
$A \to ;$	$A.pocet = 0$

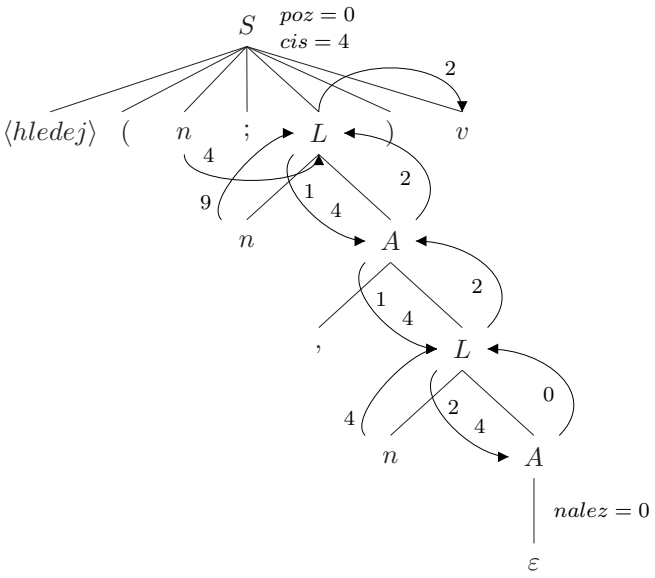
Atribut $v.vysl$ je dědičný, získává hodnotu vypočtenou v atributu $L.pocet$.



Příklad 5.12

Vytvoříme atributovou gramatiku s příkazem se dvěma argumenty. První argument je číslo, druhý argument je seznam čísel, účelem gramatiky je zjistit, zda se dané číslo nachází v seznamu a na jaké pozici (hledáme první výskyt čísla v seznamu). První prvek seznamu má index 1, pokud se hledané číslo v seznamu nenachází, je vrácen index 0.

Navrhneme překladovou gramatiku popisující syntaxi daného překladu (jediným výstupním terminálem je v, v jeho atributu $v.vysl$ bude výsledný index).



$S \to \langle hledej \rangle\,(n; L)\,v$
$L \to nA$
$A \to , L \mid \varepsilon$

Použijeme tyto atributy: $n[lex]$, $L[poz, cis, nalez]$, $A[poz, cis, nalez]$, $v[vysl]$.

Na obrázku je derivační strom výrazu $\langle hledej \rangle\,(4; 9, 4)$. Výpočet (prohledávání stromu) bude probíhat takto:

- v prvním pravidle inicializujeme atribut $L.cis$ hodnotou zjištěnou z prvního parametru (před středníkem), atribut $L.poz$ na 0,
- po derivačním stromě dolů pošleme dědičné atributy:
  - cis určující hledané číslo, tento atribut se už dál nebude měnit,
  - poz určující pozici momentálně zpracovávaného prvku seznamu, tento atribut se zvyšuje o 1 s každým dalším prvkem (v pravidle $L \to nA$),
- to, zda je hledaný prvek nalezen, bude zachyceno v syntetizovaném atributu $nalez$, který je inicializován v listu stromu (terminálním pravidle $A \to \varepsilon$), prohledávání proto probíhá od konce seznamu (od nejpravějšího listu derivačního stromu),
- pokud je nalezen prvek s danou hodnotou, změní se hodnota atributu $nalez$ na index tohoto prvku.

$S \to \langle \text{hledej} \rangle\,(n;\ \{L.poz = 0, L.cis = n.lex\}\ L)\,v\ \{v.vysl = L.nalez\}$
$L \to n\ \{A.poz = L.poz + 1, A.cis = L.cis\}$
$\quad A\ \{\text{if } n.lex = A.cis \text{ then } L.nalez = A.poz \text{ else } L.nalez = A.nalez\}$
$A \to ,\ \{L.poz = A.poz, L.cis = A.cis\}\ L\ \{A.nalez = L.nalez\}$
$A \to \varepsilon\ \{A.nalez = 0\}$

Protože používáme dědičné atributy, je nutné odlišit, kdy se které sémantické pravidlo vyhodnotí. Všechna sémantická pravidla jsou proto umístěna dovnitř syntaktických pravidel a obklopena složenými závorkami.

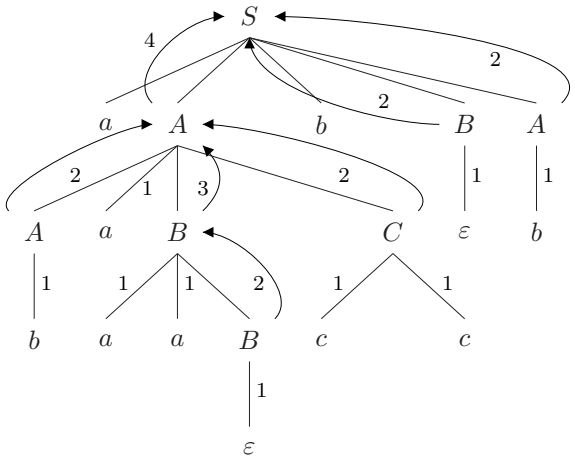
Při jakémkoliv překladu je někdy dobré znát hloubku rekurze běžného výpočtu, je důležitá zejména pro odhad časové a prostorové složitosti překladu. Pokud zvolíme implementaci metodou rekurzivního sestupu, je to jedna z mála možností, jak si udělat představu o optimálnosti procesu překladu.

Příklad 5.13

Zjistíme hloubku rekurze zpracování výrazu pro danou bezkontextovou gramatiku.

$S \to aAbB \mid aA$
$A \to AaBC \mid b$
$B \to aaB \mid \varepsilon$
$C \to aB \mid cc$

Na derivačním stromě o hloubce 4 si můžeme udělat představu o směru toku dat. Použijeme syntetizovaný atribut, který je inicializován v terminálních pravidlech na hodnotu 1 a v ostatních patrech zvyšován o 1.



Pokud je v pravidle více neterminálů (tj. v derivačním stromě z tohoto uzlu vychází více různých podstromů s potenciálně různými hloubkami), vybereme nejvyšší hodnotu ze všech, které do tohoto uzlu přicházejí, k tomu využijeme binární sémantickou funkci max.

Použijeme atributy $S[hloub],\ A[hloub], B[hloub],\ C[hloub]$. Výsledek bude uložen v syntetizovaném atributu $S.hloub$. V některých pravidlech je více než jeden výskyt téhož neterminálu, proto tyto výskyty odlišíme indexy, aby bylo možné určit, který je použit na konkrétním místě v sémantickém pravidle.

$S \to aAbB \quad \{S.hloub = \max(A.hloub, B.hloub) + 1\}$
$S \to aA \quad \{S.hloub = A.hloub + 1\}$
$A_0 \to A_1 aBC \quad \{A_0.hloub = \max(\max(A_1.hloub, B.hloub), C.hloub) + 1\}$
$A \to b \quad \{A.hloub = 1\}$
$B_0 \to aaB_1 \quad \{B_0.hloub = B_1.hloub + 1\}$
$B \to \varepsilon \quad \{B.hloub = 1\}$
$C \to aB \quad \{C.hloub = B.hloub + 1\}$
$C \to cc \quad \{C.hloub = 1\}$

5.4.3 Atributové gramatiky pro deterministický překlad výrazů

Při běžných překladech se atributy nejvíce používají při práci s výrazy, ať už matematickými, logickými nebo jakýmikoliv jinými. Proto se podíváme na $LL(1)$ a silnou $LR(1)$ atributovou gramatiku počítající matematické výrazy.

Příklad 5.14

Navrhneme $LL(1)$ atributovou gramatiku přiřazující hodnotu výrazu do proměnné. Jako základ použijeme tuto bezkontextovou $LL(1)$ gramatiku:

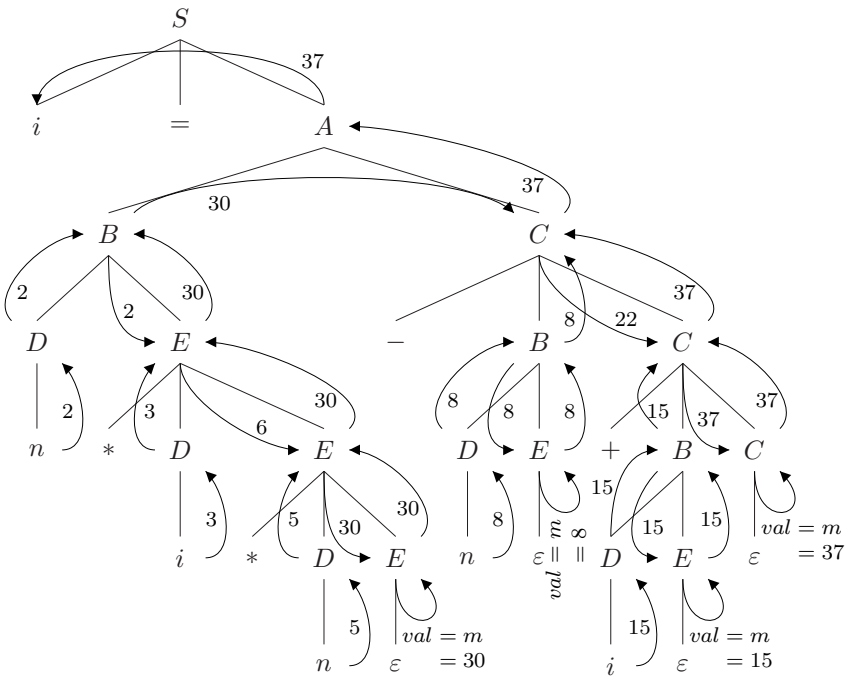
$S \to i = A$
$A \to BC$
$C \to +BC \mid -BC \mid \varepsilon$
$B \to DE$

$E \to *DE \mid /DE \mid \varepsilon$
$D \to n \mid i \mid (A)$

Kdybychom volili sémantická pravidla stejně jako v gramatice z příkladu 5.9 na straně 129, tedy výpočet pomocí syntetizovaných atributů směrem zdola nahoru, dostali bychom se do problémů s komutativitou operátorů $-$ a $/$. Například ve výrazu $3 - 2 + 1$ musíme nejdřív vypočíst podvýraz $3 - 2$, a teprve potom přičíst jedničku, jinak bychom dostali špatný výsledek. Operátory se stejnou prioritou tedy musíme počítat zleva doprava, což (téměř) znemožňuje použít pouze syntetizované atributy.

Použijeme proto jinou metodu – *ukládání mezivýsledku*. V podstromě budeme zleva doprava dolů počítat mezivýsledek v dědičném atributu m, který pak obratem v ε-pravidle pošleme v syntetizovaném atributu val nahoru.

Na obrázku 5.3 je postup vyhodnocení atributů pro výraz $X = 2*Y*5-8+Z$, v tabulce symbolů jsou právě proměnným přiřazeny tyto hodnoty: $Y = 3,\ Z = 15$. „Duální“ zasílání dat (mezivýsledek a celkový výsledek větve) se týkají pouze pravidel s neterminály C (pro + a –) a E (pro * a /). Správně bychom ještě měli ošetřit dělení nulou, to necháme na čtenáři.



Obrázek 5.3: Tok hodnot v derivačním stromě pro $LL(1)$ gramatiku

$S \to i = A\ \{\text{Uloz}(i.nazev, A.val)\}$
$A \to B\ \{C.m = B.val\}\ C\ \{A.val = C.val\}$
$C \to +B\ \{C_1.m = C_0.m + B.val\}\ C\ \{C_0.val = C_1.val\}$
$C \to -B\ \{C_1.m = C_0.m - B.val\}\ C\ \{C_0.val = C_1.val\}$

$C \to \varepsilon \ \{C.val = C.m\}$
$B \to D \ \{E.m = D.val\} \ E \ \{B.val = E.val\}$
$E \to *D \ \{E_1.m = E_0.m * D.val\} \ E \ \{E_0.val = E_1.val\}$
$E \to /D \ \{E_1.m = E_0.m/D.val\} \ E \ \{E_0.val = E_1.val\}$
$E \to \varepsilon \ \{E.val = E.m\}$
$D \to n \ \{D.val = n.lex\}$
$D \to i \ \{D.val = \text{ZjistiHodnotu}(i.nazev)\}$
$D \to (A) \ \{D.val = A.val\}$

Příklad 5.15

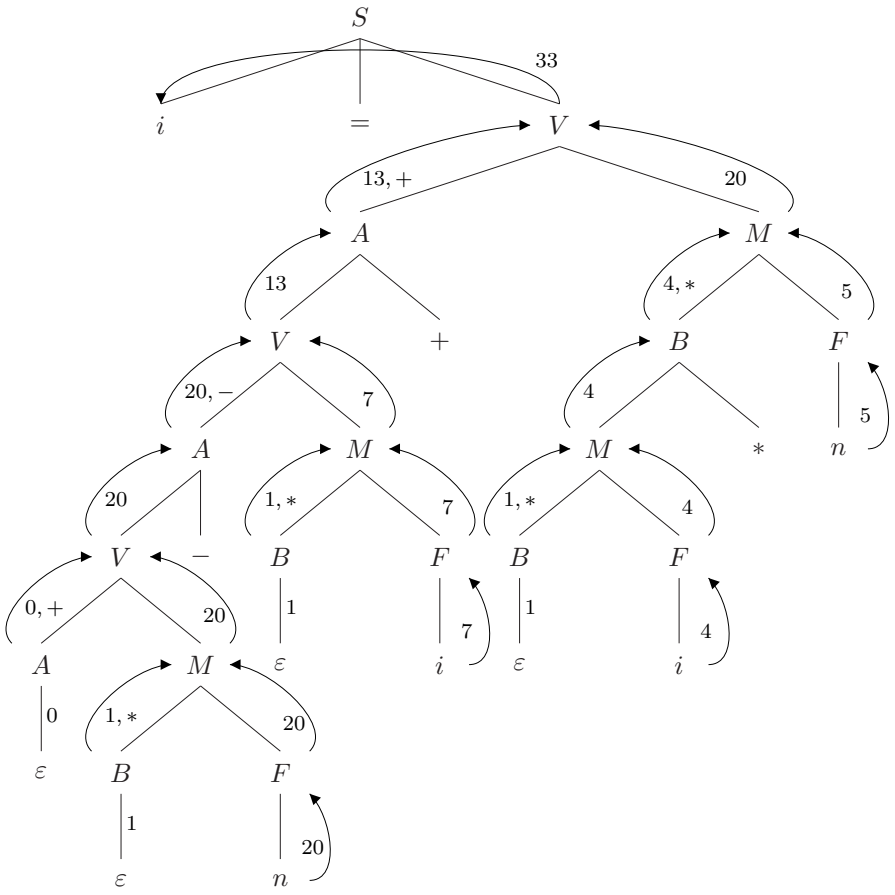
Navrhneme silnou $LR(1)$ atributovou gramatiku přiřazující hodnotu výrazu do proměnné. Použijeme následující silnou $LR(1)$ gramatiku (čtenář si může sám ověřit, že je silná $LR(1)$):

$S \to i = V$
$V \to AM$
$A \to V+ \mid V- \mid \varepsilon$
$M \to BF$
$B \to M* \mid M/ \mid \varepsilon$
$F \to n \mid i \mid (V)$

Protože při programování překladu metodou zdola nahoru není snadné implementovat dědičné atributy, použijeme pouze syntetizované a hodnoty budeme posílat v derivačním stromě zdola nahoru. Oproti LL gramatikám zde musíme odlišně řešit rozdíly mezi různými operátory stejné priority – v některých pravidlech posíláme směrem nahoru (tedy v pravidla doleva) také informaci o typu operátoru. Ošetření chyby dělení nulou opět necháváme na čtenáři.

Na obrázku 5.4 vidíme postup vyhodnocování atributů v derivačním stromě pro výraz $Vysl = 20 - X + Y * 5$, v tabulce symbolů jsou proměnným přiřazeny hodnoty $X = 7, \ Y = 4$.

$S \to i = V \ \{\text{Uloz}(i.nazev, V.val)\}$
$V \to AM \ \{\text{if } A.op = \text{S\_PLUS then } V.val = A.val + M.val \text{ else } V.val = A.val - M.val\}$
$A \to V + \ \{A.val = V.val, A.op = \text{S\_PLUS}\}$
$A \to V - \ \{A.val = V.val, A.op = \text{S\_MINUS}\}$
$A \to \varepsilon \ \{A.val = 0, A.op = \text{S\_PLUS}\}$
$M \to BF \ \{\text{if } B.op = \text{S\_MUL then } M.val = B.val * F.val \text{ else } M.val = B.val/F.val\}$
$B \to M * \ \{B.val = M.val, B.op = \text{S\_MUL}\}$
$B \to M / \ \{B.val = M.val, B.op = \text{S\_DIV}\}$
$B \to \varepsilon \ \{B.val = 1, B.op = \text{S\_MUL}\}$
$F \to n \ \{F.val = n.lex\}$
$F \to i \ \{F.val = \text{ZjistiHodnotu}(i.nazev)\}$
$F \to (V) \ \{F.val = V.val\}$



Obrázek 5.4: Tok hodnot v derivačním stromě pro silnou $LR(1)$ gramatiku

Jak v derivačním stromě, tak i v pravidlech gramatiky si můžeme všimnout odlišnosti zpracování ε-pravidel – zatímco u pravidla $A \to \varepsilon$ posíláme směrem nahoru po stromě hodnoty $+$ a 0, u pravidla $B \to \varepsilon$ to jsou hodnoty $*$ a 1. Jako číslo je vždy zvolen neutrální prvek pro danou operaci (což je například u sčínání 0).

5.5 Implementace atributového překladu

Při programování atributového překladu rozšíříme metody syntaktické analýzy, které jsme probírali v kapitolách 3.5.5 na straně 68 ($LL(1)$ překlady) a 3.7.4 na straně 89 (překlad silných $LR(k)$ gramatik).

Rozšíření spočívá v přidání sémantických pravidel (naprogramujeme sémantické funkce, ke zpracování syntaxe přidáme volání těchto funkcí a nastavení atributů) a stanovení způsobu předávání atributů mezi pravidly.

5.5.1 $LL(1)$ atributové gramatiky

Použijeme metodu rekurzivního sestupu. Stejně jako dříve u syntaktického překladu, i zde budeme pro jednotlivé neterminály programovat shodně nazvané funkce (procedury).

Zatímco v předchozích kapitolách jsme si vystačili pouze s jednou funkcí (procedurou) pro zpracování terminálů, pokud rozlišujeme vstupní a výstupní terminály, potřebujeme dvě funkce – `expect_in` pro vstupní terminály a `expect_out` pro výstupní terminály.

Atributy, které je nutno předávat v rámci pravidla z jedné strany na druhou, přenášíme obvykle jako parametry funkcí. Různé typy atributů budeme zpracovávat takto:

- *syntetizované atributy neterminálů* posíláme rekurzí směrem nahoru, pro ně použijeme parametry volané odkazem, aby bylo možné v nadřízené funkci zjistit hodnotu vypočtenou ve volané podřízené funkci,
- *dědičné atributy neterminálů* posíláme rekurzí směrem dolů, pro ně použijeme parametry volané hodnotou,
- *syntetizované atributy vstupních terminálů* ukládáme do globální proměnné (jsou součástí proměnné typu `TSymbol`), obvykle je zde ukládá lexikální analyzátor,
- *dědičné atributy výstupních terminálů* buď přímo vypisujeme, nebo ukládáme do vhodné globální proměnné (nebo dynamického seznamu či souboru).

Vstupní terminály nemají žádné dědičné atributy, výstupní terminály nemají žádné syntetizované atributy.

Pokud se některý atribut zpracovává v rámci pravé strany pravidla, použijeme lokální proměnnou v dané funkci.

Postup si ukážeme na příkladu.

Příklad 5.16

Naprogramujeme výpočet matematického výrazu podle této $LL(1)$ atributové gramatiky:

$S \to Av \ \{v.vysl = A.val, \text{VypisText}(v.vysl)\}$
$A \to B \ \{C.m = B.val\} \ C \ \{A.val = C.val\}$
$C \to +B \ \{C_1.m = C_0.m + B.val\} \ C \ \{C_0.val = C_1.val\}$
$C \to -B \ \{C_1.m = C_0.m - B.val\} \ C \ \{C_0.val = C_1.val\}$
$C \to \varepsilon \ \{C.val = C.m\}$
$B \to D \ \{E.m = D.val\} \ E \ \{B.val = E.val\}$
$E \to *D \ \{E_1.m = E_0.m * D.val\} \ E \ \{E_0.val = E_1.val\}$
$E \to /D \ \{E_1.m = E_0.m/D.val\} \ E \ \{E_0.val = E_1.val\}$
$E \to \varepsilon \ \{E.val = E.m\}$
$D \to n \ \{D.val = n.lex\}$
$D \to i \ \{D.val = \text{ZjistiHodnotu}(i.nazev)\}$
$D \to (A) \ \{D.val = A.val\}$

Výstupní symbol v určuje akci výpisu do textového okna, při jeho zpracování voláme sémantickou funkci provádějící tento výpis.

Používáme atributy $S[\,]$, $A[val]$, $B[val]$, $C[m, val]$, $D[val]$, $E[m, val]$, $n[lex]$, $i[nazev]$.

Předpokládejme, že tabulku symbolů generuje už lexikální analyzátor. Protože nepoužíváme uživatelsky definované datové typy a proměnné jsou pouze celočíselné, budou v tabulce symbolů jen proměnné, a to vždy název a hodnota. Všechny proměnné jsou (lexikálním analyzátorem) inicializovány na 0.

Ve vstupních symbolech (typu `TSymbol`) je u identifikátoru (tj. proměnné) místo řetězce s názvem pouze ukazatel do tabulky symbolů (prvky typu `TObjekt`).

Použijeme tyto deklarace:

```
type
  TTypSymbolu = (S_ID, S_NUM, S_PLUS, S_MINUS, S_MUL, S_DIV,
    S_LZAV, S_RZAV, S_VYSTUP, S_ENDOFFILE);

  TSymbol = record
    typ:        TTypSymbolu;
    atribcislo: integer;              // celé číslo (S_NUM)
    atribstr:   ↑TObjekt;             // název proměnné (S_ID)
  end;

var
  symbol:   TSymbol;
  ...       // další proměnné a datové typy, jako v příkladu 3.17 na straně 65
```

Pro rutinní práci se symboly použijeme funkci `VypisTyp(typ: TTypSymbolu)` vracející řetězcovou reprezentaci datového typu, a funkci `VypisHodn(sym: TSymbol)`, která atribut symbolu převede na řetězec podle jeho typu. Funkce `error` slouží k výpisu chybové hlášky včetně její pozice ve zdrojovém kódu:

```
procedure error(const hlaska: string);
begin
  Konec := true;
  writeln('Chyba při syntaktické analýze na řádku ',znak.cislo,
    ', sloupci ',znak.pozice,': ',hlaska);
end;
```

Pro práci s terminálními symboly definujeme tyto procedury:

```
procedure expect_in(vstupni_term: TTypSymbolu);    // vstupní symboly
begin
  if vstupni_term = symbol.typ then Lex
    else error('symbol ',VypisHodn(symbol),' není očekávaného typu ',
      VypisTyp(vstupni_term));
end;
```

```
procedure expect_out(vystupni_term: TSymbol);      // výstupní symboly
begin
  if vystupni_term.typ = S_VYSTUP
    then VypisText(vystupni_term.atribcislo);
  // chyba se nepředpokládá, případný problém se zápisem na výstup řešíme jinde
end;
```

Pokud by existovalo více výstupních terminálů, bylo by možné řešit proceduru třeba pomocí `case` (`switch`). Další funkce (procedury) programujeme podle pravidel pro jednotlivé neterminály (vždy jsou uvedena pravidla pro neterminál a pak kód procedury). Hodnoty atributů posíláme v parametrech funkcí, pro tok dat uvnitř jedné funkce využijeme lokální proměnné.

$S \rightarrow Av \ \{v.vysl = A.val, \mathrm{VypisText}(v.vysl)\}$

```
procedure S;
var pom_sym: TSymbol;
begin
  if symbol.typ in [S_NUM,S_ID,S_LZAV] then begin
    A(pom_sym);
    pom_sym := S_VYSTUP;   // výstupní terminál 'v'
    expect_out(pom_sym);
  end else error('symbol ',VypisHodn(symbol),' není očekávaného typu ',
    VypisTyp(S_NUM),', ',VypisTyp(S_ID),' nebo ',VypisTyp(S_LZAV));
end;
```

$A \rightarrow B \ \{C.m = B.val\} \ C \ \{A.val = C.val\}$

```
procedure A(var val: integer);
var pomval: integer;
begin
  if symbol.typ in [S_NUM,S_ID,S_LZAV] then begin
    B(pomval);
    C(pomval,val);
  end else error('symbol ',VypisHodn(symbol),' není očekávaného typu ',
    VypisTyp(S_NUM),', ',VypisTyp(S_ID),' nebo ',VypisTyp(S_LZAV));
end;
```

$C \rightarrow +B \ \{C_1.m = C_0.m + B.val\} \ C \ \{C_0.val = C_1.val\}$
$C \rightarrow -B \ \{C_1.m = C_0.m - B.val\} \ C \ \{C_0.val = C_1.val\}$
$C \rightarrow \varepsilon \ \{C.val = C.m\}$

```
procedure C(m: integer; var val: integer);
var pomval: integer;
begin
  case symbol.typ of
    S_PLUS: begin
      expect_in(S_PLUS);
```

```
      B(pomval);
      C(m + pomval,val)
    end;
    S_MINUS: begin
      expect_in(S_MINUS);
      B(pomval);
      C(m - pomval,val)
    end;
    S_RZAV,S_ENDOFFILE: val := m;
    else error('symbol ',VypisHodn(symbol),' není očekávaného typu ',
      VypisTyp(S_PLUS),', ',VypisTyp(S_MINUS),', ',VypisTyp(S_RZAV),
      ' nebo konec vstupu')
  end;
end;
```

$B \rightarrow D\ \{E.m = D.val\}\ E\ \{B.val = E.val\}$

```
procedure B(var val: integer);
var pomval: integer;
begin
  if symbol.typ in [S_NUM,S_ID,S_LZAV] then begin
    D(pomval);
    E(pomval,val);
  end else error(...);  // ošetření chyby podobně jako v předchozí proceduře,
end;                     // totéž platí i dále
```

$E \rightarrow *D\ \{E_1.m = E_0.m * D.val\}\ E\ \{E_0.val = E_1.val\}$
$E \rightarrow /D\ \{E_1.m = E_0.m/D.val\}\ E\ \{E_0.val = E_1.val\}$
$E \rightarrow \varepsilon\ \{E.val = E.m\}$

```
procedure E(m: integer; var val: integer);
var pomval: integer;
begin
  case symbol.typ of
    S_MUL: begin
      expect_in(S_MUL);
      D(pomval);
      E(m * pomval,val)
    end;
    S_DIV: begin
      expect_in(S_DIV);
      D(pomval);
      if pomval = 0 then error('dělení nulou')
        else E(m / pomval,val)
    end;
    S_PLUS,S_MINUS,S_RZAV,S_ENDOFFILE: val := m;
    else error(...)
  end;
end;
```

$D \to n \ \{D.val = n.lex\}$
$D \to i \ \{D.val = \text{ZjistiHodnotu}(i.nazev)\}$
$D \to (A) \ \{D.val = A.val\}$

```
procedure D(var val: integer);
begin
  case symbol.typ of
    S_NUM: begin
      val := symbol.atribcis;  // musíme zachytit předem, pak bude
      expect_in(S_NUM)         // přepsáno lexikálním analyzátorem
    end;
    S_ID: begin
      val := symbol.atribstr↑.hodnota.i;   // z tabulky symbolů
      expect_in(S_ID)
    end;
    S_LZAV: begin
      expect_in(LZAV);
      A(val);
      expect_in(RZAV)
    end;
    else error(...)
  end;
end;
```

Hlavní funkce celé analýzy bude následující:

```
procedure Analyza;
begin
  init;       // inicializace překladu včetně lexikálního analyzátoru
  Lex;        // přednačteme jeden symbol
  S;          // spustíme rekurzivní volání
  done;       // ukončení překladu, úklid paměti apod.
end;
```

5.5.2 Silné $LR(1)$ atributové gramatiky

Opět budeme vycházet z toho, co jsme probírali v syntaktické analýze. Použijeme metodu přepisu rozkladové tabulky. Narozdíl od LL překladu nemůžeme atributy předávat pomocí parametrů funkcí, proto je budeme ukládat do zásobníku zároveň s typy symbolů (jinou možností je použití dvou zásobníků – jednoho pro symboly a druhého pro atributy).

Do zásobníku tedy nebudeme ukládat pouze symboly, ale i atributy, a tomu přizpůsobíme i datový typ prvků zásobníkové abecedy. Navíc atributy mají být pouze syntetizované (nebo zpracovávané jen uvnitř pravidla), na to bereme zřetel při návrhu atributové gramatiky.

Příklad 5.17

Následující gramatika je silná $LR(1)$ (zvídavý čtenář si to může ověřit). Pravidla jsou očíslována, čísla pak použijeme v rozkladové tabulce.

$S' \to \#S$ ⓪
$S \to i = V \ \{Uloz(i.nazev, V.val)\}$ ①
$V \to AM \ \{\text{if } A.op = \text{S\_PLUS then } V.val = A.val + M.val$ ②
$\text{else } V.val = A.val - M.val\}$
$A \to V + \ \{A.val = V.val, A.op = \text{S\_PLUS}\}$ ③
$A \to V - \ \{A.val = V.val, A.op = \text{S\_MINUS}\}$ ④
$A \to \varepsilon \ \{A.val = 0, A.op = \text{S\_PLUS}\}$ ⑤
$M \to BF \ \{\text{if } B.op = \text{S\_MUL then } M.val = B.val * F.val$ ⑥
$\text{else } M.val = B.val/F.val\}$
$B \to M * \ \{B.val = M.val, B.op = \text{S\_MUL}\}$ ⑦
$B \to M / \ \{B.val = M.val, B.op = \text{S\_DIV}\}$ ⑧
$B \to \varepsilon \ \{B.val = 1, B.op = \text{S\_MUL}\}$ ⑨
$F \to n \ \{F.val = n.lex\}$ ⑩
$F \to i \ \{F.val = \text{ZjistiHodnotu}(i.nazev)\}$ ⑪
$F \to (V) \ \{F.val = V.val\}$ ⑫

Jsou zde atributy $V[val]$, $A[op, val]$, $M[val], B[op, val]$, $F[val]$, $n[lex], i[nazev]$. Typy atributů se projeví na datovém typu používaném pro zásobníkové symboly. Deklarujeme vše potřebné:

```
type
  TTypSymbolu = (S_ID, S_NUM, S_ROVNASE, S_PLUS, S_MINUS, // terminály
    S_MUL, S_DIV, S_LPAR, S_RPAR, S_ENDOFFILE,
    S_NS, S_NSC, S_NV, S_NA, S_NM, S_NB, S_NF, S_HASH);   // neterminály

  TSymbol = record
    typ:        TTypSymbolu;
    atribcislo: integer;
    atribstr:   ↑TObjekt;
  end;

  TSymbolZasob = record
    typ:        TTypSymbolu;
    atribcislo: integer;     // atributy val a lex
    atribstr:  ↑TObjekt;     // atribut nazev
    atribop:    TTypSymbolu; // použijeme S_PLUS, S_MINUS, S_MUL, S_DIV
  end;

var
  konec:      boolean;       // indikátor ukončení výpočtu, proveden accept
  symbol:     TSymbol;       // aktuální symbol načtený z proměnné vstup
  vrchol_zas: TSymbolZasob;  // symbol na vrcholu zásobníku
  zasobnik:   TZasobnik;     // prvky jsou typu TSymbolZasob
  ...         // další používané datové typy a proměnné
```

Další procedury budeme programovat podle tabulky symbolů, proto si ji sestavíme. Funkce FOLLOW a $BEFORE$ jsou následující:

$\text{FOLLOW}(S') = \{\$\}$ $\quad BEFORE(S') = \{\#\}$
$\text{FOLLOW}(S) = \{\$\}$ $\quad BEFORE(S) = \{\#\}$
$\text{FOLLOW}(V) = \{+, -,), \$\}$ $\quad BEFORE(V) = \{=, (\}$
$\text{FOLLOW}(A) = \{n, i, (\}$ $\quad BEFORE(A) = \{=, (\}$
$\text{FOLLOW}(M) = \{*, /, +, -,), \$\}$ $\quad BEFORE(M) = \{A\}$
$\text{FOLLOW}(B) = \{n, i, (\}$ $\quad BEFORE(B) = \{A\}$
$\text{FOLLOW}(F) = \{*, /, +, -,), \$\}$ $\quad BEFORE(F) = \{B\}$

	n	i	$=$	$+$	$-$	$*$	$/$	(	)	\$
S'										acc
S										r0
V				$push$	$push$				$push$	r1
A	r9	r9						r9		
M				r2	r2	$push$	$push$		r2	r2
B	$push$	$push$						$push$		
F				r6	r6	r6	r6		r6	r6
n				r10	r10	r10	r10		r10	r10
i			$push$	r11	r11	r11	r11		r11	r11
$=$	r5	r5						r5		
$+$	r3	r3						r3		
$-$	r4	r4						r4		
$*$	r7	r7						r7		
$/$	r8	r8						r8		
(	r5	r5						r5		
)				r12	r12	r12	r12		r12	r12
$\#$		$push$								

Vytvoříme procedury pro operace v tabulce. Nejdelší bude procedura pro redukci. Před každou částí zde uvádíme pravidlo, kterého se redukce týká, procedura je větvena podle čísel pravidel. Nepředpokládá se, že by byla volána s číslem neodpovídajícím žádnému pravidlu, i když i tento stav je možné ošetřit.

```
procedure reduce(cislo_prav: integer);
var
  SymbolZas: TSymbolZas;
  val: integer;
begin
  case cislo_prav of
```

$S \rightarrow \#S$

```
0: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // S
  Vyjmi_ze_zasobniku(vrchol_zas);              // #
  SymbolZas.typ := S_NSC;
  Pridej_do_zasobniku(SymbolZas);
end;
```

$S \rightarrow i = V \ \{\text{Uloz}(i.nazev, V.val)\}$

```
1: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // V
  val := vrchol_zas.atribcislo;
  Vyjmi_ze_zasobniku(vrchol_zas);              // =
  Vyjmi_ze_zasobniku(vrchol_zas);              // i
  // uložíme vypočtenou hodnotu do tabulky symbolů:
  vrchol_zas.atribstr↑.hodnota.i := val;
  SymbolZas.typ := S_NS;
  Pridej_do_zasobniku(SymbolZas);
end;
```

$V \rightarrow AM \ \{\text{if } A.op = \text{S\_PLUS then } V.val = A.val + M.val \text{ else } V.val = A.val - M.val\}$

```
2: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // M
  val := vrchol_zas.atribcislo;
  Vyjmi_ze_zasobniku(vrchol_zas);              // A
  SymbolZas.typ := S_NV;
  if vrchol_zas.atribop = S_PLUS then
    SymbolZas.atribcislo :=  vrchol_zas.atribcislo + val
  else if vrchol_zas.atribop = S_MINUS then
    SymbolZas.atribcislo :=  vrchol_zas.atribcislo - val
  else error('chyba v syntaxi aritmetického výrazu, nenalezen žádný
    ze symbolů ', VypisTyp(S_PLUS),' nebo ',VypisTyp(S_MINUS));
  Pridej_do_zasobniku(SymbolZas);
end;
```

$A \rightarrow V + \ \{A.val = V.val, A.op = \text{S\_PLUS}\}$

```
3: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // +
  Vyjmi_ze_zasobniku(vrchol_zas);              // V
  SymbolZas.typ := S_NA;
  SymbolZas.atribop := S_PLUS;
  SymbolZas.atribcislo := vrchol_zas.atribcislo;
  Pridej_do_zasobniku(SymbolZas);
end;
```

$A \to V -\ \ \{A.val = V.val, A.op = \text{S\_MINUS}\}$

```
    4: begin
      Vyjmi_ze_zasobniku(vrchol_zas);              // -
      Vyjmi_ze_zasobniku(vrchol_zas);              // V
      SymbolZas.typ := S_NA;
      SymbolZas.atribop := S_MINUS;
      SymbolZas.atribcislo := vrchol_zas.atribcislo;
      Pridej_do_zasobniku(SymbolZas);
    end;
```

$A \to \varepsilon\ \ \{A.val = 0, A.op = \text{S\_PLUS}\}$

```
    5: begin
      SymbolZas.typ := S_NA;
      SymbolZas.atribop := S_PLUS;
      SymbolZas.atribcislo := 0;
      Pridej_do_zasobniku(SymbolZas);
    end;
    ... // pro symboly M a B to bude podobné jako pro V a A,
        // ale nesmíme zapomenout ošetřit dělení nulou
```

$F \to n\ \ \{F.val = n.lex\}$

```
    10: begin
      Vyjmi_ze_zasobniku(vrchol_zas);              // n
      SymbolZas.typ := S_NF;
      SymbolZas.atribcislo := vrchol_zas.atribcislo;
      Pridej_do_zasobniku(SymbolZas);
    end;
```

$F \to i\ \ \{F.val = \text{ZjistiHodnotu}(i.nazev)\}$

```
    11: begin
      Vyjmi_ze_zasobniku(vrchol_zas);              // i
      with SymbolZas do begin
        typ := S_NF;
        atribcislo := vrchol_zas.atribstr↑.hodnota.i;
      end;
      Pridej_do_zasobniku(SymbolZas);
    end;
    ... // pro pravidlo F → (S) pouze předáme nahoru atribut
  end;
end;
```

Zbývají tabulkové operace `error` (ta je zároveň použita pro ošetření chyby), `push` vkládající terminály ze vstupu do zásobníku a `accept` pro přijetí vstupu.

```
procedure error(const hlaska: string);
begin
  Konec := true;
  writeln('Chyba při syntaktické analýze na řádku ',znak.cislo,
    ', sloupci ',znak.pozice,': ',hlaska);
end;

procedure push;
var
  SymbolZas: TSymbolZas;
begin
  with SymbolZas do begin
    typ := symbol.typ;
    atribcislo := symbol.atribcislo;
    atribstr := symbol.atribstr;
  end;
  Pridej_do_zasobniku(SymbolZas);
  Lex;         // lexikální analyzátor načte další symbol
end;

procedure accept;
begin
  Konec := true;
end;
```

Ještě chybí inicializace, ukončení, řídící procedura tabulky a hlavní procedura analýzy:

```
procedure Init;
var
  SymbolZas: TSymbolZas;
begin
  ...                          // inicializace vstupu a výstupu
  Vytvor_zasobnik;
  SymbolZas.typ := S_HASH;
  Pridej_do_zasobniku(SymbolZas);
  Lex;
  Konec := false;
end;

procedure Done;
begin
  Zlikviduj_zasobnik;          // uvolní paměť zabranou zásobníkem
  ...                          // uzavření vstupu a výstupu
end;

procedure Akce;
begin
  case vrchol_zas of
    S_NSC: if symbol.typ = S_ENDOFFILE then accept
      else error('očekáván konec souboru');
```

```
    S_NS: if symbol.typ = S_ENDOFFILE then reduce(0)
      else error('očekáván konec souboru');
    S_NV: case symbol.typ of
        S_PLUS,S_MINUS,S_RZAV: push;
        S_ENDOFFILE: reduce(1);
        else error('symbol ',VypisHodn(symbol),' není očekávaného typu ',
          VypisTyp(S_PLUS),', ',VypisTyp(S_MINUS),', ',VypisTyp(S_RZAV),
          ' nebo konec souboru');
      end;
    S_NA: if symbol.typ in [S_NUM,S_ID,S_LZAV] then reduce(9)
      else error('symbol ',VypisHodn(symbol),' není očekávaného typu ',
      VypisTyp(S_NUM),', ',VypisTyp(S_ID),' nebo ',VypisTyp(S_LZAV));
    ... // atd. pro všechny neterminály
    S_NUM: if symbol.typ in [S_PLUS,S_MINUS,S_MUL,S_DIV,S_RZAV,S_ENDOFFILE]
      then reduce(10)
      else error(...);        // ošetření chyby podobně jako v předchozím kódu,
    S_ID: case symbol.typ of  // to platí i dále
        S_PLUS,S_MINUS,S_MUL,S_DIV,S_RZAV,S_ENDOFFILE: reduce(11);
        S_ROVNASE: push;
        else error(...);
      end;
    S_ROVNASE: if symbol.typ in [S_NUM,S_ID,S_LZAV] then reduce(5)
      else error(...);
    ... // atd. pro všechny terminály
    else error(...);
  end;
end;

procedure Analyza;
begin
  Init;
  while (not Konec) do Akce;
  Done;
end;
```

Úkoly ke kapitole 5

1. Překladovou gramatiku z příkladu 5.3 na straně 122 obohaťte o operátory odčítání (priorita stejná jako sčítání), dělení a celočíselného dělení div (priorita operátoru stejná jako u operátoru násobení).
2. Sestrojte konečný překladový automat (přímo, bez konstrukce gramatiky), který dokáže dělit binární čísla třemi. Předpokládejte, že na vstupu je vždy číslo dělitelné třemi.

Nápověda: nejdřív vyzkoušejte dělení binárních čísel „ručně“. Zbytek po dělení je po každém kroku některé z čísel 0, 1 nebo 2, použijte stavy N, J a D pro uložení této informace. Automat reaguje na momentální stav (tj. N, J nebo D) a vstupní symbol (0 nebo 1), akcí je přechod do některého ze stavů (podle momentálního zbytku po dělení) a výpisu číslice na výstup.

3. Podle následující regulární překladové gramatiky (překládající některé prvky syntaxe PASCALU do C) sestrojte konečný překladový automat reprezentovaný δ-funkcí i přechodovou tabulkou.

   $RPG = (\{S\},\ \{\langle begin\rangle, \langle end\rangle, x\},\ \{①, ②, ⓧ\},\ R, S)$, kde x je kterýkoliv ze symbolů ve vstupní abecedě přímo neuvedených a v množině R jsou tato pravidla:
   $S \to \langle begin\rangle\ ①S \mid \langle end\rangle\ ②S \mid x ⓧ S \mid \langle end\rangle\ ②$

4. Zjistěte, zda následující překladová gramatika je $LL(1)$ – její vstupní gramatiku najdete v úkolu 4 na straně 94. Pokud je to $LL(1)$ překladová gramatika, sestrojte k ní překladový automat reprezentovaný rozkladovou tabulkou a podle této tabulky zpracujte (přeložte) jakékoliv slovo z jazyka vstupní gramatiky delší než 4.

   $PG = (\{S, A, B, C\},\ \{a, b, c, d\},\ \{ⓐ, ⓑ, ⓒ, ⓓ\},\ R, S)$, pravidla jsou

   $S \to ⓐabA ⓑ \mid cbBC ⓒ ⓑ$
   $A \to cA ⓒ \mid dB ⓑ ⓑ \mid a ⓐ$
   $B \to bB ⓑ S \mid a ⓐ ⓐ$
   $C \to cC ⓒ \mid ⓒ cB d ⓓ \mid \varepsilon$

5. Je dána překladová gramatika $PG = (\{A, B\},\ \{a, b\},\ \{x, y\},\ R,\ A)$ s pravidly v R

   $A \to aAy \mid bxB$
   $B \to bxB \mid bx$

   Zjistěte, jaký překlad generuje tato gramatika a sestrojte překladový automat reprezentovaný δ-funkcí.

6. Následující atributová gramatika generuje deklarace v syntaxi jazyka C a deklarované proměnné přidává do tabulky symbolů.

   $D \to T\ \ \{S.d\_typ = T.typ\}\ \ S;$
   $T \to \langle int\rangle\ \ \{T.typ = integer\}$
   $T \to \langle float\rangle\ \ \{T.typ = float\}$
   $T \to \langle char\rangle\ \ \{T.typ = char\}$
   $S \to i\ \ \{\text{PridejDoTab}(i.nazev, S.d\_typ),\ M.d\_typ = S.d\_typ\}\ \ M$
   $M \to ,\ \ \{S.d\_typ = M.d\_typ\}\ \ S$
   $M \to \varepsilon$

   Vytvořte derivační strom věty $\boxed{\langle int\rangle\ \ x, y, prom;}$ s vyznačením toku hodnot atributů. Zjistěte, zda je gramatika typu $LL(1)$ a promyslete si způsob implementace rekurzivním sestupem – sémantické funkce, které je třeba naprogramovat, dále parametry funkcí podle neterminálů volané hodnotou nebo odkazem a kód těchto funkcí.

7. Následující bezkontextová gramatika popisuje deklarace proměnných v PASCALovské syntaxi.

   $D \rightarrow S : T;$
   $T \rightarrow \langle int \rangle \mid \langle real \rangle \mid \langle char \rangle$
   $S \rightarrow iM$
   $M \rightarrow , S \mid \varepsilon$

   Přidejte sémantická pravidla tak, aby gramatika plnila podobnou funkci jako gramatika se syntaxí jazyka C v předchozím úkolu (tj. přidává deklarované proměnné do tabulky symbolů).

   Nápověda: protože se vstup vyhodnocuje zleva doprava, na místě uvedení názvu proměnné (terminál i) na vstupu neznáme její datový typ a proto nemůžeme do tabulky symbolů přidat oba údaje najednou; tento problém můžeme vyřešit například tak, že na místě uvedení názvu proměnné přidáme do tabulky symbolů záznam s datovým typem, který dokážeme snadno vyhledat (například $\langle nedef \rangle$), a pak v pravidle, kdy konečně zjistíme datový typ proměnné, projdeme celou tabulku symbolů a ve všech záznamech s datovým typem $\langle nedef \rangle$ tento typ zaměníme za zjištěný. Jinou možností je dočasné uschování názvů proměnných ve spojovém seznamu.

8. Promyslete si implementaci atributové gramatiky, kterou jste vytvořili v úkolu 7, pro interpretační překladač. Pro datové typy vytvořte typy symbolů stejně jako například pro identifikátory (názvy proměnných), dvojtečku, čárku a středník, pro prvky v tabulce symbolů pak použijte záznamy obsahující tyto tři údaje:

   - název proměnné (typu řetězec),
   - datový typ proměnné (typu `TTypSymbolu`),
   - hodnotu proměnné (použijte variantní záznam nebo union).

   Tabulku symbolů implementujte jako spojový seznam výše popsaných záznamů. Vytvořte přístupové funkce k tabulce (vyhledání proměnné, zjištění a změnu hodnoty proměnné, přidání nové proměnné a její odstranění, vhodným způsobem zajistěte ošetření chyb.

9. Je dána gramatika popisující syntaxi seznamu čísel:

   $S \rightarrow L$
   $L \rightarrow nA$
   $A \rightarrow , L \mid ;$

   Přidejte sémantická pravidla tak, aby

   (a) v atributu $S.soucet$ byl součet všech čísel v seznamu.
   (b) v atributu $S.max$ byla hodnota nejvyššího prvku seznamu.

(c) v atributu $n.index$ každého prvku seznamu bylo pořadové číslo tohoto prvku (začínáme zleva indexem 1).

10. Je dána následující překladová gramatika:

$Z \to Sv$
$S \to (\ A \mid ca$
$A \to a\ ) \mid [\ cSb\ ] \mid \langle\ B \mid \varepsilon$
$B \to b\ \rangle \mid S \mid \varepsilon$

Přidejte sémantická pravidla tak, aby v atributu výstupního terminálu

(a) $v.leve$ byl počet všech levých závorek ve vygenerovaném výrazu (tj. závorek (, [a $\langle$).
(b) $v.kulate$ byl počet všech kulatých závorek (levých i pravých) ve vygenerovaném výrazu.
(c) $v.n$ byl počet všech (vstupních terminálních) symbolů ve vygenerovaném výrazu, které nejsou (jakýmikoliv) závorkami, tedy symbolů a, b a c.
(d) $v.eps$ byl počet použití ε-pravidla během derivace (nezapomeňte na inicializaci syntetizovaného atributu ve všech terminálních pravidlech).

11. K dané překladové gramatice přidejte sémantická pravidla tak, aby při interpretaci docházelo k vyhodnocování logických výrazů v uvedené syntaxi.

$V \to Sv$
$S \to A\ \langle or \rangle\ S \mid A$
$A \to B\ \langle and \rangle\ A \mid B$
$B \to (S) \mid C < C$
$C \to i \mid n$

Použijte atributy $V[\,]$, $v[vysl]$, $S[log]$, $A[log]$, $B[log]$, $C[val]$, $i[name]$, $n[lex]$. Určete, které jsou dědičné a které syntetizované. Dále použijte vhodnou sémantickou funkci pro zjištění hodnoty proměnné z tabulky symbolů a pro rozhodování příkaz typu if.

12. Zjistěte, zda je daná překladová gramatika typu $LL(1)$. Pokud ano, přidejte sémantická pravidla tak, aby se funkčně shodovala s atributovou překladovou gramatikou, kterou jste vytvořili v úkolu 11 a naprogramujte metodou rekurzivního sestupu.

$V \to Sv$
$S \to AD$
$D \to \langle or \rangle\ S \mid \varepsilon$
$A \to BE$
$E \to \langle and \rangle\ A \mid \varepsilon$
$B \to (S) \mid C < C$
$C \to i \mid n$

KAPITOLA 6

Jak co naprogramovat

Předchozí kapitoly obsahovaly základ, který je nutný pro naprogramování prakticky jakéhokoliv překladače. V této kapitole se podíváme na pokročilejší věci, „speciality", které se nám někdy mohou hodit, třebaže je nevyužijeme pokaždé, když programujeme překladač.

Budeme se zabývat programováním uživatelských datových typů a proměnných, metodami zpracování složitějších výrazů s mnoha úrovněmi operátorů, strukturou programu s rekurzivním voláním procedur a událostmi, a krátce se také podíváme na generování cílového kódu v kompilátoru.

6.1 Uživatelské datové typy a proměnné

Programováním základních datových typů a proměnných jsme se zabývali již dříve v kapitole 4.1. V interpretačním překladači ukládáme hodnoty proměnných přímo do tabulky, nemusíme se starat o jejich adresaci. Ovšem v kompilačním překladači platí:

- globální proměnné jsou uloženy v datových segmentech programu a místo v paměti pro ně musí být rezervováno již při překladu (statická sémantika),
- lokálním proměnným při překladu místo nerezervujeme (rezervaci lze provést pouze v případě, kdy jazyk neumožňuje rekurzi ani vzájemná volání podprogramů), ale je nutné zajistit rezervaci místa v zásobníkovém segmentu během provádění programu, kdykoliv je funkce volána, a jejich odstranění po ukončení funkce (dynamická sémantika),
- nesmíme zapomenout na implicitní přetypování a typovou kontrolu.

Jestliže programovací jazyk dovoluje pracovat se strukturou bloků, ve kterých existují lokální proměnné, pak paměť pro tyto proměnné musíme na začátku vykonávání bloku

rezervovat také v zásobníkovém segmentu, abychom mohli po ukončení bloku rezervaci zrušit.

Dále se podíváme na reprezentaci složitějších (uživatelsky definovaných) datových typů, soustředíme se spíše na použití v interpretačním překladači.

6.1.1 Pole

Pole je chápáno jako posloupnost prvků stejného typu (homogenní datová struktura). Může jít o spojitý úsek paměti dlouhý (počet prvků pole) $*$ (délka jednoho prvku pole v B), pak jde o ryze statické pole, nebo jako ukazatel na takový úsek paměti. Druhý způsob je vhodný například tehdy, když programovací jazyk dovoluje dynamicky měnit délku pole.

U vícerozměrných statických polí volíme reprezentaci posloupností úseků paměti, tedy jakési pole polí. Například pro 2-rozměrné pole (matici) jsou prvky výsledného pole jednotlivé řádky matice[1], tyto řádky jsou opět pole. Inspirací nám mohou být dynamická vícerozměrná pole v jazyku C.

První problém, který je nutné vyřešit, je začlenění proměnné tohoto typu do tabulky symbolů. Můžeme postupovat například takto:

- u jednorozměrného pole s indexy z intervalu `M..N` a prvky datového typu `T` přidáme do tabulky symbolů nejdříve datový typ `T`, pak datový typ pole prvků typu `T` (odkaz na řádek tabulky, kde je `T` určen) s daným intervalem a potom teprve proměnnou, jejíž datový typ udáme odkazem na řádek tabulky s přidaným datovým typem; poslední členění lze vynechat, poskytuje však možnost znovupoužitelnosti pole pro více proměnných stejného datového typu,
- u vícerozměrného pole datový typ „rozložíme“ až k jednorozměrným polím, například v Pascalovské definici obecně pro datový typ

  `array [`M_1`..`N_1`, `M_2`..`N_2`, ..., `M_k`..`N_k`] of T`

  vytvoříme posloupnost datových typů

  T_1 `= array [`M_k`..`N_k`] of T`
  T_2 `= array [`M_{k-1}`..`N_{k-1}`] of `T_1
  `...`
  T_{k-1} `= array [`M_2`..`N_2`] of `T_{k-2}
  T_k `= array [`M_1`..`N_1`] of `T_{k-1}`,`

  a všechny od T_1 postupně přidáme do tabulky symbolů; v rámci optimalizace můžeme pak tyto dílčí datové typy využít i dále, pokud budou hranice indexů vyhovovat,
- intervaly `[`M_i`..`N_i`]` můžeme také pokládat za datové typy – před přidáním pole do tabulky symbolů přidáme do této tabulky datový typ `interval [`M_i`..`N_i`]`.

[1] Tento postup je platný, pokud chceme rozdělit vícerozměrné pole „po řádcích“. Kdybychom je chtěli rozdělit „po sloupcích“, typ T_1 v uváděném postupu by měl hranice indexů `[`M_1`..`N_1`]`.

V programu na místě deklarace pole je třeba tomuto poli vyhradit místo v paměti. Tady opět záleží na tom, zda vícerozměrné pole ukládáme po řádcích nebo po sloupcích. Například u pole deklarovaného

```
p: array [1..3, 7..8] of integer
```

bude při ukládání po řádcích paměť organizovaná podle tabulky 6.1.

p[1,7]	p[1,8]
p[2,7]	p[2,8]
p[3,7]	p[3,8]

➪

p[1,7]
p[1,8]
p[2,7]
p[2,8]
p[3,7]
p[3,8]

Tabulka 6.1: Reprezentace datového typu *pole* v paměti

Nejdřív je v paměti uložen celý první řádek, pak celý druhý řádek atd. Adresu prvku `p[`u, v`]` v obecné reprezentaci `p[`$\mathtt{M}_1$`..`$\mathtt{N}_1$`, `$\mathtt{M}_2$`..`$\mathtt{N}_2$`]` získáme výpočtem

$$\mathrm{adr} + \mathrm{sizeof}(int) * \Big((u - \mathtt{M}_1) * (\mathtt{N}_1 - \mathtt{M}_1) + (v - \mathtt{M}_2)\Big)$$

kde adr je adresa začátku celé datové struktury pole a $\mathrm{sizeof}(int)$ je délka datového typu `integer`.

Slovně: po přesunu na adresu celého pole se nejdřív posuneme na začátek toho řádku, na kterém se prvek nachází (vynásobíme index počítaný od 0 délkou řádku, což je $(\mathtt{N}_1 - \mathtt{M}_1)$) a potom se na řádku posuneme o druhý index opět počítaný od čísla 0. U polí, kde je syntaxí předepsán jako první index vždy 0, je výpočet samozřejmě jednodušší.

6.1.2 Záznam a struktura

Záznam (v C struktura) je nehomogenní posloupnost prvků, tzn. prvky této posloupnosti mohou být obecně různého typu. Jestliže je v kódu uvedena deklarace datového typu

```
p_typ = record
  p1: T1;
  p2: T2;
  ...
  pn: Tn;
end;
```

pak v tabulce symbolů budou již zahrnuty datové typy `T1`, `T2`, ..., `Tn`. Přidáme tedy nový datový typ pojmenovaný `p_typ`.

Při deklaraci proměnné typu záznam vyhradíme v paměti určitý úsek se strukturou odpovídající tvaru záznamu. Většina překladačů do paměti ukládá jednotlivé prvky záznamu v tom pořadí, v jakém jsou vypsány ve zdrojovém textu a na oblast odpovídající přímo velikosti datového typu, některé překladače se však pokoušejí optimalizovat rychlost přístupu do paměti přeskládáním položek nebo například ukládáním na sudé adresy nebo pro rozsáhlejší typy dat na celé paměťové stránky.

Metoda získání adresy určitého prvku záznamu závisí na skutečném tvaru záznamu v paměti. Pokud se rozhodneme ukládat prvky přímo za sebou podle zdrojového kódu

a jednotlivým prvkům vyhradíme přesně tolik místa, kolik potřebují, pak adresu i-tého prvku záznamu získáme přičtením délek všech datových typů předchozích prvků záznamu k adrese začátku záznamu – pro proměnnou `prom` datového typu `p_typ`:

$$\mathrm{adr}(\texttt{prom.pi}) = \mathrm{adr}(\texttt{prom}) + \sum_{j=1}^{i-1}(\mathrm{sizeof}(\texttt{Tj}))$$

PASCAL umožňuje používat *variantní záznamy*, jazyky vycházející z C zase *uniony*. Variantní záznam (resp. union) dovoluje více prvkům sdílet tutéž paměť. Například u deklarace

```
type
  TTypUdalosti = (uChyba, uKlavesnice, uMys, ...);

  TUdalost = record
    vlastnik: TObject;             // komu je událost určena
    case typ: TTypUdalosti of
      uChyba: (kod: integer);      // kód chyby, která nastala
      uKlavesnice: (
        klavesa: integer;          // která klávesa byla stisknuta
        znak: char;                // znaková klávesa: její kód, jinak 0
        shft, ctrl, alt: boolean); // držena některá z˜uvedených kláves
      uMys: (tlacitko: integer;    // která tlačítka myši jsou stisknuta
        pozx, pozy: integer);      // pozice myši na obrazovce
      ...
  end;
```

bude v paměti vyhrazeno místo (předpokládejme, že pracujeme v 32-bitovém systému a tedy datový typ `integer` zabírá 4B):

$$\begin{aligned}\mathrm{sizeof}(\texttt{TUdalost}) &= \mathrm{sizeof}(\texttt{TObject}) + \mathrm{sizeof}(\texttt{TTypUdalosti}) + \\ &\quad + \max(4, 4+1+1+1+1, 4+4+4) \\ &= \mathrm{sizeof}(\texttt{TObject}) + \mathrm{sizeof}(\texttt{TTypUdalosti}) + 12 \qquad (6.1)\end{aligned}$$

Sečetli jsme délky datových typů vnitřních proměnných `vlastnik` a `typ`, a potom jsme k nim přičetli maximální hodnotu součtů délek datových typů použitých pro různé větve příkazu `case`.

Číslo vypočtené podle vzorce (6.1) je délka datového typu. Pokud se potřebujeme dostat na určité paměťové místo, musíme také mít na paměti variantnost dat. Například jestliže chceme vědět, zda je stisknuta klávesa CTRL, použijeme tento postup:

$$\mathrm{adr} + \mathrm{sizeof}(\texttt{TObject}) + \mathrm{sizeof}(\texttt{TTypUdalosti}) + \mathrm{sizeof}(\texttt{integer}) + 1 + 1$$

kde adr je adresa začátku celého záznamu. Délku datových typů `char` a `boolean` předpokládáme 1.

6.1.3 Třída a objekt

V objektově orientovaných jazycích se setkáváme s třídami a objekty. Třída je obdoba datového typu záznam, obsahuje však nejen datové prvky, ale také metody (členské funkce), příp. vlastnosti. Objektem rozumíme instanci třídy, tedy je to obdoba proměnné.

Pro implementaci tříd a objektů neexistuje žádná šablona, hodně záleží na míře objektovosti jazyka. V nejjednodušším případě můžeme metodu do záznamu zahrnout jako adresu funkce, zpětnou vazbu zajistíme označením funkce za metodu objektu přímo v jejím těle nebo hlavičce a přidáme ukazatel na objekt, se kterým metoda v této chvíli pracuje. Adresa funkce v záznamu nám říká, kde najít kód funkce–metody, ukazatel na aktivní objekt funkci řekne, kde má hledat lokální data a metody objektu, ke kterým potřebuje mít přístup.

Metoda musí být odlišena od funkcí, které nejsou metodami, aby nemohlo dojít k jejímu volání mimo specifikaci a kontext objektu.

Objektové jazyky obvykle implementují dědičnost a polymorfismus, což přináší nutnost používání virtuálních nebo přepisovaných metod. Zde je třeba vytvořit pro každý objekt s virtuálními metodami tabulku virtuálních metod (VMT), kde určíme, která „fyzická“ metoda se ve skutečnosti má provést, jestliže uživatel zavolá metodu s určitým jménem. Kdykoliv je pak použita metoda s určitým názvem, program si ve VMT zjistí adresu skutečné metody.

6.2 Vlastní paměťový model

Každý proces má operačním systémem přiřazenu paměť. Na začátku této paměti je identifikace procesu (hlavička) s daty důležitými především pro operační systém. Část těchto dat bývá namapována nebo uložena v systémové části paměti v tabulce procesů. Následuje segment kódu, kde je uložen programový kód procesu (získaný ze spustitelného souboru), který je postupně operačním systémem prováděn. Následují segmenty paměti, které využívá samotný proces.

Jestliže používání našeho programovacího jazyka spočívá především v práci s pamětí (to se týká také databázových systémů) nebo chceme dát uživateli do rukou částečnou možnost ovládání paměti třeba pomocí ukazatelů, stojí za úvahu implementace správy paměti.

Vhodný paměťový model vybereme podle potřeb zdrojového jazyka, můžeme použít některý z modelů probíraných v předmětu *Operační systémy*. Obvykle stačí tyto části paměti:

- datová oblast pro tabulku symbolů (alespoň její globální část, která se nemění za běhu procesu),
- stack (zásobník), jestliže zdrojový jazyk překladače dovoluje rekurzivní volání funkcí,
- heap (halda, dynamická oblast), jestliže zdrojový jazyk překladače dovoluje používání dynamických datových typů a ukazatelů.

V moderních operačních systémech je používána virtualizace adres a odkládací prostor (ať už soubor nebo nebo celá disková oblast), tedy není problém s množstvím paměti. Je nutné pouze rezervovat dostatečné místo při spuštění programu (konkrétní hodnoty mohou být například určeny v konfiguračním souboru překladače nebo předány jako volitelný parametr při spuštění překladače) a naprogramovat správu této paměťové oblasti.

Programy pracující s velkým množstvím dat používají také vlastní obdobu odkládacího souboru, aby nezahlcovaly operační paměť. Pro správu vlastního odkládacího souboru můžeme využít metody známé z předmětu *Operační systémy*.

6.3 Interpretace výrazů

Výrazy můžeme interpretovat více způsoby:

1. Rekurzivním rozkladem výrazu podle priority operátorů – vyžaduje neustálé prohledávání řetězce a několikanásobné zpracování téhož symbolu, neoptimální.
2. Atributovou gramatikou a její implementací – popsáno v kapitole 5.4.3.
3. Výraz nejdřív převedeme do postfixu (vhodnou překladovou gramatikou) a pak interpretujeme atributovou gramatikou – gramatika pro překlad do postfixu je v příkladu 5.10 na straně 131, zpracování postfixu je naznačeno v kapitole 4.2.3. Tento postup je výhodný zejména pro LR překlad.
4. Použijeme typ překladu, který není bezkontextový (dva zásobníky).

Čtvrá metoda se nedá naprogramovat technikami uvedenými v předchozích kapitolách, ale přesto se jeví efektivní zejména pro výrazy se složitou strukturou priorit operátorů a její velkou výhodou je snadná rozšiřitelnost pro další operátory a jejich úrovně priorit.

6.3.1 Použití dvou zásobníků

Tuto metodu můžeme použít v kombinaci s některou z výše uvedených. Protože při LR (silné) analýze obecně nelze použít syntetizované atributy, které naopak u interpretace matematických výrazů jsou velmi užitečné, můžeme pro reprezentaci celého programu použít silnou LR gramatiku, ve které matematické výrazy zpracováváme „zvlášť" metodou dvou zásobníků. U LL překladu se zase takto lze vyhnout používání atributu pro mezivýsledek.

Potřebujeme tedy dva zásobníky – jeden na hodnoty (například čísla) a druhý na operátory. Každý operátor může být reprezentován jedním znakem, pak se jedná o zásobník prvků typu `char`, nebo použijeme výčtový typ.

Výpočet probíhá takto:

1. Lexikální analyzátor musí rozeznat unární mínus od binárního – u operátoru mínus rozpoznáme unaritu jednoduše tak, že unární se vyskytuje vždy pouze buď na za-

čátku výrazu, nebo za závorkou nebo za jiným operátorem (pokud to definice jazyka umožňuje).

2. Stanovíme prioritu operátorů (například posloupnost množin P_1, P_2, ..., v množině P_1 budou operátory s nejvyšší prioritou, v P_2 s prioritou o něco nižší atd.).

3. Průběh výpočtu:

   - jestliže je na vstupu hodnota, uložíme ji do zásobníku hodnot, u proměnné zjistíme její hodnotu a tu uložíme,
   - pokud je na vstupu operátor, pak v zásobníku operátorů vybíráme prvky a vyhodnocujeme, dokud tyto operátory mají prioritu menší nebo rovnu prioritě zpracovávaného operátoru nebo dokud nenarazíme na dno – pak zpracovávaný operátor uložíme na vrchol zásobníku,
   - operátory vybírané ze zásobníku operátorů vyhodnocujeme tak, že ze zásobníku hodnot vybereme patřičný počet hodnot (podle toho, zda jde o unární, binární, ternární, ... operátor) a tyto hodnoty použijeme jako operandy (pozor na pořadí, nejdřív vybereme poslední operand, první operand vybereme jako poslední),
   - levou (otevírací) závorku pouze uložíme do zásobníku operátorů,
   - když ve vstupu najdeme pravou (uzavírací) závorku, pak ze zásobníku operátorů vybíráme a vyhodnocujeme operátory tak dlouho, dokud nenarazíme na levou závorku; pravou závorku už do zásobníku neukládáme.

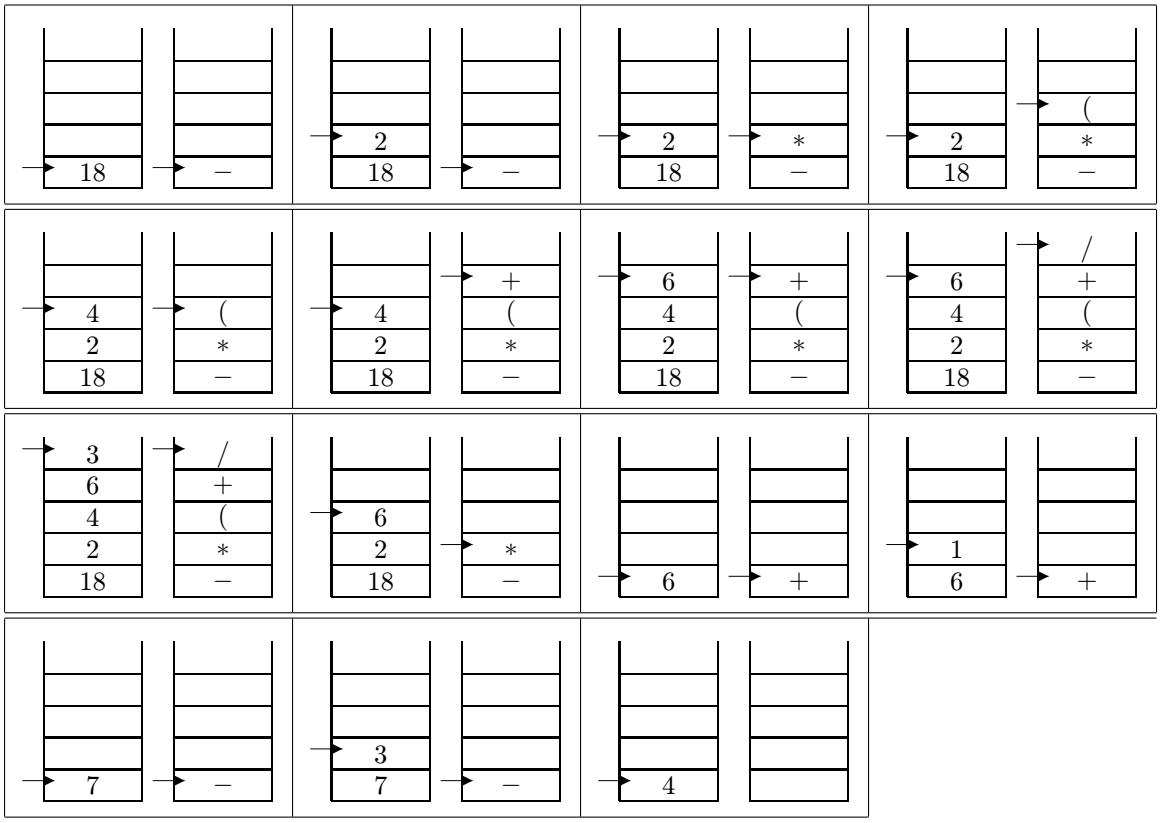
4. Po načtení posledního prvku výrazu vyhodnotíme postupně všechny operátory, které v zásobníku operátorů zbyly a potom by v zásobníku hodnot měla být právě jedna hodnota, která je výsledkem celého výpočtu.

Příklad 6.1

Postup ukážeme na výpočtu výrazu $18 - 2 * (4 + 6/3) + 1 - 3$.

Obsah zásobníků v jednotlivých krocích výpočtu vidíme v tabulce 6.2 na straně 160. První buňka tabulky zachycuje stav po načtení druhého symbolu výrazu, operátoru $-$. V dalších buňkách jsou symboly zatím jen ukládány do zásobníků, protože k vyhodnocení operátoru má dojít vždy až tehdy, když je do zásobníku ukládán operátor s nižší prioritou (pak je třeba vyhodnotit postupně všechny operátory se stejnou nebo vyšší prioritou z vrcholu zásobníku), a také když je načtena pravá závorka (druhá buňka na třetím řádku).

V poslední buňce celé tabulky je stav zásobníků po ukončení výpočtu – zásobník hodnot obsahuje výsledek a zásobník operátorů je prázdný.



Tabulka 6.2: Interpretace výrazu automatem se dvěma zásobníky

6.3.2 Implementace

Implementaci ukážeme na příkladu (je programován v DELPHI). Chyby jsou ošetřeny pomocí výjimek.

Příklad 6.2

Budeme pracovat s výrazy nad tímto jazykem:

- aritmetické operátory $+, -, *, /$ (operátor pro odčítání může být unární i binární),
- relační operátory $<, >, <=, >=, =, <>$,
- logické operátory `AND`, `OR`, `NOT`,
- celá čísla, reálná čísla, pravdivostní hodnoty (konstanty `TRUE`, `FALSE`),
- identifikátory představující názvy proměnných,
- pomocné symboly – závorky.

Nejvyšší prioritu mají operátory $*$ a $/$, následují $+$ a $-$ včetně unárního mínus, nižší prioritu mají relační operátory, po nich operátor `NOT` a nejnižší prioritu mají zbylé logické operátory.

Deklarujeme potřebné datové typy a proměnné:

```
type
  TTypSymbolu = (S_ID, S_NUMINT, S_NUMFLT, S_BOOL, S_ENDOFINPUT,
    S_PLUS, S_MINUS, S_UMINUS, S_MUL, S_DIV, S_LZAV, S_RZAV,
    S_LESS, S_GT, S_LQ, S_GQ, S_EQ, S_NEQ, S_NOT, S_AND, S_OR);

  TMnozinaOp  = set of TTypSymbolu;

  TSymbol = record
  case typ: TTypSymbolu of
    S_NUMINT: (i: integer);
    S_NUMFLT: (f: float);
    S_BOOL:   (b: boolean);
    S_ID:     (hodn: ↑TObjekt);        // ukazatel do tabulky symbolů,
  end;                                 // viz kap. 4.1.3

  EInvalidFormat = class(Exception);   // výjimky - ošetření chyb
  EInvalidType   = class(Exception);

const
  Prior1 = [S_MUL,S_DIV];                      // aritmetické, priorita 1
  Prior2 = [S_PLUS,S_MINUS,S_UMINUS];          // aritmetické, priorita 2
  Prior3 = [S_LESS,S_GT,S_LQ,S_GQ,S_EQ,S_NEQ];// relační
  Prior4 = [S_NOT];                            // logický  NOT
  Prior5 = [S_AND,S_OR];                       // logické  AND, OR
  Unarni = [S_UMINUS,S_NOT];                   // unární

var
  symbol: TSymbol;
  zas_h:  TZasobnikHodnot;    // zásobník, do kterého ukládáme čísla
  zas_op: TZasobnikOperatoru; // zásobník, do kterého ukládáme operátory
```

Zásobníky jsou programovány jako třídy (resp. objekty tříd). Pro práci se zásobníky nám slouží metody `Vyjmi(hodnota)` a `Pridej(hodnota)`, první z těchto metod vrací `false`, pokud je zásobník prázdný. Další metodou je `Prazdny` vracející `true`, pokud je zásobník prázdný. V kódu používáme tyto funkce:

```
procedure Lex;
function NaText(t: TTypSymbolu): string;
procedure ZjistiHodnotuZTabulky(var sym: TSymbol);
  // do symbolu uloží hodnotu proměnné (na vstupu je symbol typu S_ID)
procedure Plus(arg1, arg2: TSymbol; var vysl: TSymbol);
procedure Minus(arg1, arg2: TSymbol; var vysl: TSymbol);
procedure Krat(arg1, arg2: TSymbol; var vysl: TSymbol);
procedure Deleno(arg1, arg2: TSymbol; var vysl: TSymbol);
function Mensi(arg1, arg2: TSymbol): boolean;
function Vetsi(arg1, arg2: TSymbol): boolean;
function Rovno(arg1, arg2: TSymbol): boolean;
```

Funkce pro aritmetické a relační operace vnitřně provádějí typovou kontrolu a také přetypování, když je to nutné. V případě chybného datového typu nebo matematické chyby (např. dělení nulou) vyvolají výjimku.

Naprogramujeme proceduru, která pouze „technicky“ zpracuje operátor, který dostane jako parametr. Tuto proceduru voláme vždy, když ze zásobníku vyjmeme operátor.

```
procedure ZpracujOperator(op: TTypSymbolu);
var h1, h2, vysl: TSymbol;
begin
  if (not zas_h.Vyjmi(h1)) then
    raise EInvalidFormat.Create(NaText(op) + ': Operand nenalezen');
            // Jestliže je zásobník hodnot prázdný, vyvoláme výjimku, která
            // ukončí tuto funkci, funkci, ze které byla volána, ..., až do
            // místa, kde je výjimka ošetřena konstrukcí
            //  try
            //    ...   příkazy, také ten, který vyvolal tuto funkci
            //  except
            //    on EInvalidFormat do .... ošetření chyby
            //    .... další výjimky pro chyby, které mohly nastat
            //  end;

  if (not (op in Unarni)) then
    if (not zas_h.Vyjmi(h2)) then
      raise EInvalidFormat.Create(NaText(op) + ': První operand nenalezen');
  vysl.typ := h1.typ;
  case op of
    S_UMINUS: case h1.typ of
      S_NUMINT: vysl.i := -h1.i;
      S_NUMFLT: vysl.r := -h1.r;
      else raise EInvalidType.Create('Unární mínus je jen pro čísla.');
    end;
    S_PLUS: Plus(h2, h1, vysl);    // také provedou případné přetypování
    S_MINUS: Minus(h2, h1, vysl);
    S_MUL: Krat(h2, h1, vysl);
    S_DIV: Deleno(h2, h1, vysl);
    S_LESS: vysl.b := Mensi(h2, h1);
    S_GT: vysl.b := Vetsi(h2, h1);
    S_LQ: vysl.b := Mensi(h2, h1) or Rovno(h2, h1);
    S_GQ: vysl.b := Vetsi(h2, h1) or Rovno(h2, h1);
    S_EQ: vysl.b := Rovno(h2, h1);
    S_NEQ: vysl.b := not Rovno(h2, h1);
    S_AND: Krat(h2, h1, vysl);
    S_OR: Plus(h2, h1, vysl);
    S_NOT: if (h1.typ = S_BOOL) then vysl.b := not h1.b
      else raise EInvalidType.Create('NOT je jen pro hodnoty Ano-Ne.');
    else raise EInvalidFormat.Create(NaText(op)+': operátor špatně umístěn.');
  end;
  zas_h.Pridej(vysl);
end;
```

Následující proceduru voláme, když ze vstupu načteme operátor a je potřeba ze zásobníku vyjmout a vyhodnotit všechny operátory se stejnou nebo vyšší prioritou než operátor, který byl právě načten. Procedura postupně vybírá operátory ze zásobníku a zpracovává je, dokud jsou z množiny předané jako druhý parametr procedury. Zastaví se tedy v okamžiku, kdy narazí na operátor s nižší prioritou.

```
procedure ZpracujPodlePriority(op: TTypSymbolu; operatory: TMnozinaOp);
var op2: TTypSymbolu;
begin
  while not zas_op.Prazdny do begin
    zas_op.Vyjmi(op2);
    if (op2 in operatory) then ZpracujOperator(op2)
    else begin
      zas_op.Pridej(op2);   // vrátíme zpátky
      break;
    end;
  end;
  zas_op.Pridej(op);
end;
```

Zbývá ještě hlavní procedura syntaktické analýzy výrazu, která provede vyhodnocení výrazu. Procedura `VyhodnotVyraz()` postupně volá proceduru `Lex()` a zpracovává symboly, výsledek uloží do globální proměnné `symbol`.

```
procedure VyhodnotVyraz(var vysledek: THodnota);
var operator: TTypSymbolu;
begin
  zas_h  := TZasobnikHodnot.Create;
  zas_op := TZasobnikOperatoru.Create;
  Lex;        // přednačteme jeden symbol
  try
    while not (symbol.typ = S_ENDOFINPUT) do begin
      case symbol.typ of
        S_NUMINT,S_NUMFLT,S_BOOL: zas_h.Pridej(symbol);
        S_ID: begin
          ZjistiHodnotuZTabulky(symbol);
          zas_h.Pridej(symbol);
        end;
        S_LZAV: zas_op.Pridej(S_LZAV);
        S_RZAV: while (not zas_op.Prazdny) do begin
            zas_op.Vyjmi(operator);
            if (operator = S_LZAV) then break;
            ZpracujOperator(operator);
          end;
        end else begin
          operator := symbol.typ;
          if (operator in Prior1) then
            ZpracujPodlePriority(operator, Prior1)
```

```
          else if (operator in Prior2) then
            ZpracujPodlePriority(operator, Prior1+Prior2)
          else if (operator in Prior3) then
            ZpracujPodlePriority(operator, Prior1+Prior2+Prior3)
          else if (operator in Prior4) then
            ZpracujPodlePriority(operator, Prior1+Prior2+Prior3+Prior4)
          else if (operator in Prior5) then
            ZpracujPodlePriority(operator, Prior1+Prior2+Prior3+Prior4+Prior5)
          else raise EInvalidType.Create(
            'Místo očekávaného operátoru je ' + NaText(symbol.typ));
        end;
      end;  // case
      Lex;
    end;  // while
    while (not zas_op.Prazdny) do begin
      zas_op.Vyjmi(operator);
      ZpracujOperator(operator);
    end;
    zas_h.Vyjmi(symbol);
    if (not zas_h.Prazdny) then
      raise EInvalidFormat.Create('Nevhodné ukončení výrazu.');
  finally
    zas_h.free;
    zas_op.free;
  end;  // try - finally
end;
```

Velkou výhodou metody se dvěma zásobníky je použitelnost i pro jazyk, který není $LL(1)$. To se týká také jazyka výrazů z příkladu 6.2, ve kterém lze kombinovat aritmetické, relační a logické operátory[2].

6.4 Interpretace příkazů, událostí a podprogramů

6.4.1 Příkazy

Posloupnost příkazů v interpretačním překladači lze samozřejmě reprezentovat a zpracovávat tak, jak bylo uvedeno v předchozích kapitolách (tj. bez použití intermediálního kódu, pomocí atributové gramatiky). Když ovšem máme interpretační konverzační překladač, který je pevně spojen s grafickým prostředím, je ještě další možnost.

Program se skládá z příkazů, přesněji každá funkce obsahuje posloupnost příkazů. Tyto příkazy na sebe navazují (podle principu von Neumanna), tedy pro interpretaci je dobré

[2]Tento jazyk není $LL(1)$, protože při LL překladu nelze rozlišit závorky příslušející aritmetickému podvýrazu a závorky logického výrazu. Například v řetězci $((x < 1)\text{and}(y > 2))\text{or}(z \leq x)$ náleží první levá závorka logickému výrazu, ale druhá levá závorka aritmetickému podvýrazu, což při jejich načtení není patrné.

vytvořit dynamickou strukturu podobnou sémantickému stromu z kapitoly 4.2.2 na straně 107 reprezentovanou dynamickým větveným seznamem. Práce s pointery se vyznačuje především vyšší rychlostí zpracování, což je u interpretace považováno za velkou výhodu.

Narozdíl od dříve popsaných metod zde průběžně generujeme intermediální kód ve formě sémantického stromu.

Příklad 6.3

Nadefinujeme tvar příkazu pro interpretační konverzační překladač. Programujeme v prostředí DELPHI.

Překladač je programován formou „Model–View“, tedy oddělíme vnitřní (Model) a grafickou (View) reprezentaci programu, ale vzájemně je propojíme. V grafickém prostředí je kód (příkazy) zachycen stromem podobným, jaký se používá také k zobrazení struktury adresářů (složek), každý uzel v tomto stromě představuje jeden příkaz. Uzly jsou pak napojeny na prvky datové struktury příkazů typu `TPrikaz`, která je po spuštění programu přímo interpretována. Pro lepší představu – prostředí může být podobné jako na obrázku 1.2 na straně 14 (tam šlo o grafický editor programů pro děti).

Každý příkaz má svůj typ, implicitní parametry, které uživatel může v grafickém prostředí nastavovat dle svého uvážení, v grafickém prostředí je také možné měnit typ příkazu (při změně se vždy nastaví implicitní parametry pro nový typ příkazu).

Typy proměnných jsou celé číslo, reálné číslo, pravdivostní hodnota a řetězec. Proměnné řadíme do seznamu proměnných, jeho naprogramování necháváme na čtenáři (může to být například dynamický seznam nebo obdobně navržená třída, lze také přidat metody pro práci se seznamem). Můžeme dokonce vytvořit dvě varianty – jednu tříděnou pro běžný seznam proměnných ve funkci, druhou netříděnou pro seznam parametrů funkce.

```
type
  TTypHodnoty = (PROM_INT, PROM_FLT, PROM_BOOL, PROM_STR);
  TNazevPromenne = string[20];

  THodnota = record
    nazev: TNazevPromenne;
    case typ: TTypHodnoty of
      PROM_INT:  (i: integer);
      PROM_FLT:  (f: float);
      PROM_BOOL: (b: boolean);
      PROM_STR:  (s: ↑string);  // řetězec je dynamicky alokován
  end;

  TPromenna = class
  public
    nazev:   TNazevPromenne;
    hodnota: THodnota;
    procedure Assign(p: TPromenna);
  end;

  TPromSeznam = ......          // seznam proměnných
```

```
TTypPrikazu = (
  P_PRAZDNY, P_ZADEJ_PRIKAZ,   P_VOLEJ_FUN,     P_KONEC_FUN,
  P_WHILE,   P_FOR,            P_KONEC_SMYCKY,  P_KONEC_PROG,
  P_IF,      P_BLOK,           P_ZADEJ_VYRAZ,   P_ZMEN_RYCHLOST,
  P_WAIT,    P_CEKEJ_TLACITKO, P_CEKEJ_KLAVESA, P_CEKEJ_MYS,  ...);

TPrikaz = class
private         // formální parametry jsou pro každý příkaz známy
  FTyp: TTypPrikazu;
  parametry: TPromSeznam;    // netříděný seznam skutečných parametrů
  ...
  procedure ImplicitniParametry;
  function GetParam(const Index: integer): TPromenna;
  procedure SetParam(const Index: integer; Value: TPromenna);
  procedure SetTyp(Value: TTypPrikazu);
  ...
public
  constructor Create(t: TTypPrikazu);
  destructor Destroy; override;
  procedure Assign(const prikaz: TPrikaz);
  procedure AssignParams(const par: TPromSeznam);
  ...
  property Typ: TTypPrikazu read FTyp write SetTyp;
  property Param[const Index:integer]: TPromenna read GetParam write SetParam;
  ...
end;

PUzel = ↑TUzel;
TUzel = record
  prikaz: TPrikaz;          // příkaz spojený s tímto uzlem stromu
  next, prev, cont: PUzel;  // následující a předchozí uzel a vnořené příkazy
  Node: TTreeNode;          // odkaz na grafickou reprezentaci uzlu
end;
```

Příkazy jsou zřetězeny ve spojovém seznamu v uzlech typu `TUzel`. Každý uzel má odkaz na předchozí příkaz v seznamu `prev` (ve stromové struktuře odkaz směrem nahoru), následující příkaz `next` (odkaz směrem dolů) a na případný blok vnořených příkazů `cont` (odkaz směrem doprava na „podsložky“), který využijeme například u příkazu typu `P_WHILE`. U příkazu `P_IF` nedefinujeme větev „else“, tedy také stačí jediný odkaz na vnitřní příkazy.

K příkazům lze přistupovat především přes vlastnosti `Typ` a `Param`. Pokud se uživatel v grafickém prostředí rozhodne změnit typ příkazu v daném uzlu, stačí (po příslušných změnách v grafickém prostředí) do první vlastnosti přiřadit nový typ. Vnitřně je zavolána procedura `SetParam`, která změní typ (`FTyp`) a zavolá proceduru `ImplicitniParametry`.

Obdobně funguje změna parametrů. Při změně v grafickém prostředí se nová hodnota parametru přiřadí do vlastnosti `Param`.

6.4.2 Příkazy větvení

U příkazů větvení (`if`, `case`, `switch` apod.) rozlišujeme větve posloupností příkazů, které se mají vyhodnocovat, od větví příkazů, které se nemají vyhodnocovat. U interpretace je problém především v tom, že lexikální a syntaktická analýza by měla být provedena v obou typech větví, ale sémantická (rozhodně její dynamická část) a interpretace pouze v prvním typu. Pokud se v neinterpretovaných větvích nacházejí dynamické sémantické chyby, neměly by mít vliv na překlad, například příkaz

```
if x<>0 then vysl := y/x
        else vysl := y/(x+1);
```

je naprosto v pořádku, třebaže v první větvi by potenciálně mohlo dojít k sémantické chybě dělení nulou. Protože však díky testování je první větev vykonávána jen tehdy, když je proměnná `x` nenulová, k sémantické chybě nedojde[3].

Tento problém lze řešit několika způsoby závisejícími především na použité metodě interpretace.

Jestliže používáme intermediální kód ve formě sémantického stromu, který pak interpretujeme (kapitola 4.2.2 na straně 107), pak jednoduše řídíme průchod stromem podle výsledků podmínek v uzlech, ve kterých dochází k větvení.

Pokud jsme však zvolili interpretaci „zabudovanou“ do atributové gramatiky ve formě sémantických pravidel, existuje také poměrně jednoduchý způsob, jak rekurzívně určovat, zda daný příkaz má být vyhodnocen. Použijeme dědičný atribut (pojmenovaný třeba $provest$ nebo $prov$) nabývající jedné z pravdivostních hodnot $true$ nebo $false$, který u prvního pravidla gramatiky nastavíme na $true$ a posíláme po derivačním stromě směrem dolů ke všem pravidlům generujícím příkazy jazyka.

Ve vnořeném příkazu větvení je tento atribut samozřejmě také poslán dolů, ale pouze ve větvích, které se mají vyhodnotit, s hodnotou „zděděnou“ od nadřízeného uzlu, a do větví, které nemají být vyhodnoceny, je odeslán s hodnotou $false$.

Při vyhodnocení příkazu zjistíme, jakou hodnotu má tento atribut. Pokud $true$, pak provedeme příslušná sémantická pravidla (například vyhodnotíme výraz, vykreslíme okno nebo vypíšeme řetězec na výstup), v opačném případě bude jediným provedeným sémantickým pravidlem odeslání záporné hodnoty „prováděcího“ atributu po derivačním stromě dolů (v případě, kdy tento příkaz obsahuje volání dalších příkazů, třeba příkaz bloku, cyklu nebo rozhodování).

Ukázka použití atributu $prov$ je v souhrnném příkladu v příloze A.

[3]Pokud však je proměnná `x` v paměti sdílené několika procesy nebo vlákny, pak pravděpodobnost sémantické chyby není nulová; může se totiž teoreticky stát, že mezi testováním na (ne)nulovost a provedením příkazu v první větvi jiné vlákno či proces nastaví tuto proměnnou na nulu. Pak je třeba použít synchronizační mechanismy, které jsme probírali v předmětu *Operační systémy*.

6.4.3 Příkazy cyklů

U příkazů cyklů řešíme jeden důležitý problém – zpravidla je nutné vracet se v kódu. Návrat v kódu (zdrojovém nebo některém interním) lze provést dvěma způsoby v závislosti na tom, jak je naprogramován lexikální analyzátor – rozlišíme použití pro jednoprůchodový a víceprůchodový překladač.

1. Lexikální a syntaktický analyzátor jsou v různých průchodech. Jestliže je lexikální analyzátor v samostatném průchodu a tedy syntaktický analyzátor má k dispozici posloupnost symbolů podle celého zdrojového souboru (dynamický seznam nebo soubor), provádí návrat samotný syntaktický analyzátor a blok kódu uvnitř cyklu včetně podmínky je vyhodnocován lexikální analýzou pouze jednou.

Případ řešíme zachycením ukazatele do seznamu výstupu lexikální analýzy. Sestavíme pravidlo a kód pro příkaz `while`. Předpokládáme, že z neterminálu P jsou generovány všechny příkazy zdrojového programovacího jazyka včetně příkazu `while`. Syntaktické pravidlo je $P \rightarrow wMdP$, kde terminály w a d představují klíčová slova while–do, z neterminálu M se generuje logický výraz podmínky a P je vnořený příkaz. Pravidlo atributové gramatiky a příslušný úsek kódu v proceduře `P` vypadá takto:

$P_0 \rightarrow w\ \{M.prov = P_0.prov,\ \text{UlozUmisteni},\ \text{repeat},\ \text{NactiUmisteni}\}$
$\quad Md\ \{P_1.prov = P_0.prov \text{ and } M.bool\}\ \ P_1\ \{\text{until not}(P_1.prov \text{ and } M.bool\}$

```
S_WHILE: begin
  expect(S_WHILE);
  pom_sym := zpracovavany_symbol;      // ukazatelé (pointery),
  repeat                               // zachycují momentální pozici v kódu
    zpracovavany_symbol := pom_sym;    // vrátíme se na zachycenou pozici
    M(prov,b);
    expect(S_DO);
    P(prov and b);
  until not (prov and b);
end;
```

2. Lexikální a syntaktický analyzátor jsou ve společném průchodu. Syntaktický analyzátor nemá přístup k celému svému zdroji najednou, proto musí provést navracení lexikální analyzátor, a to přímo v zdrojovém souboru. Můžeme postupovat například takto:

- kdykoliv lexikální analyzátor narazí na klíčové slovo určující cyklus, třeba `while`, uloží svou pozici – vytvoří „zarážku“,
- když syntaktický analyzátor narazí na konec cyklu, vyhodnotí, zda se má tento cyklus znovu provést:
  - ano – požádá lexikální analyzátor o návrat k nejbližší zarážce (tj. od nejvnitřnějšího cyklu),
  - ne – požádá lexikální analyzátor o zrušení nejbližší (právě poslední) zarážky,

u příkazu, kde podmínku uvádíme před tělem cyklu (včetně příkazu `while`), je ovšem nutné provést navracení po ukončení každého cyklu a při negativním vyhodnocení podmínky odskočit na první příkaz za cyklem,

- je třeba ošetřit i takové cykly, které se ve skutečnosti neprovedou ani jednou (to se může stát u cyklů typu `while`).

Nevýhodou je nutnost provádět lexikální analýzu obsahu cyklu a podmínky opakovaně pro všechny průchody. To se sice dá řešit vytvořením „dočasného mezikódu" vnitřku cyklu, ale toto řešení není zrovna transparentní, už proto, že cyklus může být třeba v rozsahu celého programu a také cykly mohou být navzájem vnořené, čímž ztrácíme výhodu zařazení obou fází překladu do společného průchodu.

Tento případ řešíme vytvořením funkcí pro komunikaci s lexikálním analyzátorem s tím, že je nutné zajistit samotné navracení ve vstupu (soubor by mohl být načten jako stream, ve kterém se lze snadněji pohybovat).

6.4.4 Podprogramy

Také v interpretovaných jazycích bývá možné vytvořit vlastní funkce nebo procedury, obecně podprogramy. Pro reprezentaci můžeme zvolit dynamický seznam, jehož prvky představují jednotlivé podprogramy, jedním z prvků je pak také hlavní program (hlavní funkce). Některé podprogramy můžeme také označit za událostní, tedy sloužící k ošetření událostí klávesnice, myši, práce s tlačítky, ...

Rozlišujeme však reprezentaci podprogramu v kódu (statická syntaxe a sémantika) a pak během samotné interpretace (dynamická sémantika). Zde musíme počítat také s tím, že v podprogramu obecně lze volat jiný podprogram a také je možné používat rekurzi. Jako nejvhodnější se jeví níže uvedená reprezentace.

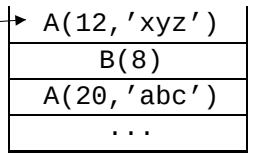
Při interpretaci je podprogram plně určen strukturou (záznamem) nazývanou obvykle *aktivační záznam*. Tato struktura obsahuje všechny dynamické (měnící se) informace:

- hodnoty lokálních proměnných, jde o tabulku symbolů pro tento blok (podprogram je také blokem),
- skutečné parametry (dosazené za formální parametry funkce či procedury), které při spuštění interpretace zařadíme do lokální tabulky symbolů (viz předchozí bod),
- odkaz na kód (příkazy) podprogramu,
- ukazatel na příkaz, který je právě vykonáván, (do kódu v předchozím bodu), atd.

Při samotné interpretaci pak používáme *zásobník* (stack), do kterého ukládáme aktivační záznam právě zpracovávané funkce. Jsou v něm tedy uloženy aktivační záznamy všech podprogramů, které jsou právě ve zpracování, podprogram se záznamem níže byl pozastaven, když zavolal podprogram, jehož záznam je nyní výše.

Zásobník aktivačních záznamů nemusí být programován zvlášť – do aktivačního záznamu podprogramu jednoduše umístíme odkaz na aktivační záznam podprogramu, jímž byl vyvolán, nejnižší aktivační záznam má tento odkaz nastaven na null (nil).

Na obrázku 6.1 je zjednodušený nákres struktury zásobníku aktivačních záznamů pro případ, kdy nejdřív je volána funkce `A` s parametry `20` a `'abc'`, uvnitř této funkce voláme funkci `B` a dále uvnitř funkce `B` je volána nepřímou rekurzí funkce `A`, tentokrát s jinými parametry:



Obrázek 6.1: Zásobník aktivačních záznamů

```
A(20,'abc')
  B(8)
    A(12,'xyz')
```

Můžeme postupovat například takto:

1. Na začátku interpretace vložíme do zásobníku aktivační záznam hlavní funkce nebo jinak zajistíme, aby se program po vyhodnocení jiných volaných funkcí vrátil do hlavní funkce. Zde bývají uloženy globální proměnné.
2. Jestliže je volána funkce F, vytvoříme její aktivační záznam, inicializujeme v něm skutečné parametry, s jakými byla funkce volána, nastavíme ukazatele na první příkaz této funkce apod. Pak tento záznam přidáme na vrchol zásobníku a vyhodnocujeme funkci F.
3. Po ukončení vyhodnocování funkce F zajistíme případné načtení návratové hodnoty funkce na určité místo v programu (navracení – F byla volána z funkce, jejíž aktivační záznam je v zásobníku další na řadě), pak vyjmeme její aktivační záznam ze zásobníku, uvolníme paměť, kterou tento záznam zabíral a v práci programu pokračujeme vyhodnocováním podle následujícího aktivačního záznamu v zásobníku.
4. Vyjmutím posledního aktivačního záznamu ze zásobníku končí činnost celého programu.

Příklad 6.4

Sestavujeme aplikaci ve Windows, využíváme možností grafického rozhraní. Navážeme na kód uvedený v příkladu 6.3 na straně 165.

Zároveň s vytvářením programu uživatelem se provádí lexikální, syntaktická a částečně i sémantická analýza (parametry příkazů uživatel zadává pomocí dialogových oken, tedy lze okamžitě kontrolovat jejich počet a datový typ, pokud by v určitém okamžiku nemělo být možné zadat některý příkaz, pak jeho tlačítko je jednoduše znepřístupněno, syntaktickou strukturu programu vlastně tvoří editor podle pokynů uživatele, apod.).

Každá funkce je reprezentována dynamickým stromem, jehož uzly jsou příkazy. U příkazů větvení (`IF`, `CASE`) se uzel větví, u složených příkazů (např. tělo příkazu `WHILE`)

představuje první větev tělo příkazu, druhá pak uzel s příkazem následujícím po tomto příkazu. Tato struktura připomíná sémantický strom v kapitole 4.2.2 na straně 107.

Při interpretaci postupujeme shora dolů a zleva doprava. V uzlech je stanoveno, které větve mají být interpretovány (to je důležité například u podmíněných příkazů `IF` a `CASE`), jak je naznačeno v sekci 6.4.2.

Aktivační záznam, funkci a program definujeme takto:

```
type
  TProgram = class;   // dopředné deklarace
  TFunkce = class;

  TAktivacniZaznam = class
  public
    FNaslednik: TAktivacniZaznam;  // následník v zásobníku aktivačních záznamů
    FFunkce:  TFunkce;     // ukazatel na funkci, které patří tato struktura
    FAktivni: PUzel;       // ukazatel na aktivní příkaz této funkce
    FNavrat:  TPromenna;   // návratová hodnota (odkaz)
    FProm:  TPromSeznam;   // skutečné lokální proměnné, pro funkce s parametry,
    ...                    // jsou zde uloženy také skutečné parametry
    constructor Create;
    destructor Destroy;
    ...
    property Prom[const Index: TNazevPromenne]: TPromenna read GetProm
                                                          write SetProm;
  end;

  TStrom = class      // slouží jako rodičovská třída pro funkce
  private
    procedure SetAktivni(Value: PUzel);
    procedure SetHodnota(Value: TPrikaz);
    function GetHodnota: TPrikaz;
  protected
    FPrvni: PUzel;          // kořen stromu
    FAktivni: PUzel;
    procedure ZrusVetev(var uzel: PUzel); virtual;
    procedure ZrusUzel(var uzel: PUzel);  virtual;
    ...
  public
    constructor Create; virtual;
    destructor Destroy; override;
    procedure Insert(prikaz: TPrikaz);         // přidá nový uzel za aktivní
    procedure InsertBefore(prikaz: TPrikaz);   // přidá nový před aktivní
    procedure Remove;                          // odstraní aktivní uzel
    property Aktivni: PUzel read FAktivni write SetAktivni default nil;
    property Prvni: PUzel read FPrvni;
    property Hodnota: TPrikaz read GetHodnota write SetHodnota;
  end;
```

```
TFunkce = class (TStrom)
private
  FNazev: string;
  procedure SetNazev(const Value: string);
  function GetProm(const Index: TNazevPromenne): TPromenna;
  procedure SetProm(const Index: TNazevPromenne; Value: TPromenna);
  ...
public
  AZ: TAktivacniZaznam;        // struktura zachycující momentální stav funkce
  FormPromenne: TPromSeznam;   // formální proměnné
  FormParametry: TPromSeznam;  // formální parametry
  Navrat: TTypHodnoty;         // návratový typ funkce
  ...
  constructor Create(AOwner: TProgram); virtual;
  destructor Destroy;                   override;
  procedure Init(var fi: TFceInfo);   // provede se při volání funkce,
    // fi obsahuje skutečné parametry a odkaz na místo návratu hodnoty
  procedure Done;                     // provede se při ukončení funkce
  function VykonejPrikaz(var az: TAktivacniZaznam): TTypPrikazu;
    // vykoná aktivní příkaz, posune se na další
  property Nazev: string read FNazev write SetNazev;   // název funkce
  ...
end;

TSeznamFunkci = .....         // vhodně implementujeme seznam funkcí

TProgram = class
private
  FAktivniAZ: TAktivacniZaznam;    // aktivní aktivační záznam funkce,
  ...                              // která je právě vyhodnocována
public
  FFunkce: TSeznamFunkci;   // prvky: TFunkce, používá se při návrhu funkcí
  Udalosti: TUFronta;       // fronta událostí
  ...
  constructor Create;
  destructor Destroy;
  procedure Init;       // voláme, když chceme spustit sestavený program
  procedure Done;       // úklid po skončení programu, připraví další spuštění
  procedure VolaniFunkce(var az: TAktivacniZaznam);
    // vytvoří nový aktivační záznam, zařadí do zásobníku a spustí vyhodnocení
  function SestavFunkciUdalosti(var az: TAktivacniZaznam;
    const u: TUdalost): boolean;         // ošetření událostí:
  procedure UdalostKeyUp(Sender: TObject; var Key: Word; Shift: TShiftState);
  procedure UdalostKeyPress(Sender: TObject; var Key: char);
  procedure UdalostMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
  procedure UdalostButtonClick(Sender: TObject);
  procedure CekejNaUdalost(tu: TTypUdalosti);
  procedure Delay(sec: integer);
  ...
end;
```

Implementaci jednotlivých metod zde nebudeme uvádět, závisí na určení překladače. Jen je třeba pamatovat na zajištění přístupu k aktivačním záznamům funkcí, které jsou v zásobníku hlouběji pod záznamem právě prováděné funkce, aby bylo možné pracovat s ne-lokálními proměnnými. Z aktivačního záznamu dané funkce je možné pracovat s jejími proměnnými prostřednictvím vlastnosti `Prom`, která představuje jakési pole (ne skutečné, pouze jako rozhraní), jehož indexy jsou názvy proměnných.

6.4.5 Události

V příkladu 6.4 jsme v definici třídy `TProgram` viděli náznak zpracování událostí. Události je třeba „odchytávat“, zjišťovat, komu jsou určeny, a zajistit předání a včasné vyhodnocení.

Program může být *řízen událostmi*. To znamená, že na začátku programu provedeme určitou akci (ošetření události „Začátek programu“, kterou lze chápat jako hlavní funkci programu), a dále ve smyčce čekáme, dokud se ve *frontě událostí* neobjeví některý prvek. Pak tuto událost ošetříme a opět ve smyčce čekáme na další.

Implementaci můžeme nechat na uživateli našeho překladače nebo ji zahrneme do definice jazyka a uživatel pouze určí funkci, která se provede, pokud k dané události dojde. Proces zpracování programu představuje práci s frontou událostí. U každé události určíme vlastníka (komu je určena nebo kde vznikla) a stanovíme, která funkce se má provést.

Příklad 6.5

Naprogramujeme implementaci událostí doplňující předchozí příklady. Každá událost má svého vlastníka, a pak ve variantním záznamu data podle typu události.

```
type
  TTypUdalosti = (uNic, uKlavesnice, uMys, uTimer, uTlacitko, ...);
  TUdalost = record
    vlastnik: TObject;   // pak přetypujeme podle typu vlastníka události
    case typ: TTypUdalosti of
      uKlavesnice: ( _klavesa: integer;
                     _znak: char;       // pokud = #0, nejde o znakovou klávesu
                     _shift, _ctrl, _alt: boolean );
      uMysUp:      ( _tlacitko, _pozx, _pozy: integer );
      uMysDown:    ( _tlacitko, _pozx, _pozy: integer );
      uTimer:      ( );
      uTlacitko:   ( _t_akce: (akce_stisk, akce_uvolneni) );
      ...
  end;

  PUPrvek = ↑TUPrvek;
  TUPrvek = record             // prvek fronty událostí
    udalost: TUdalost;
    next: PUPrvek;
  end;
```

```
  TUFronta = class                    // fronta událostí
  private
    prvni, posledni: PUPrvek;         // začátek a konec spojového seznamu
    FPocet: integer;                  // počet událostí ve frontě
    function GetPrazdny: boolean;
  public
    constructor Create;
    destructor Destroy;
    procedure Pridej(u: TUdalost);
    function Dej(var u: TUdalost): boolean;
    procedure VyprazdniFrontu;
    property Prazdny: boolean read GetPrazdny;
    property Pocet: integer read FPocet;
  end;
```

Vlastníkem události je například tlačítko, které bylo stisknuto, nebo objekt, na kterém došlo ke stisknutí tlačítka myši nebo klávesy na klávesnici.

Implementace závisí na programovacím jazyce, ve kterém překladač vytváříme. Jedním z možných řešení je rozdělení řízení událostí na dvě asynchronní části:

1. Sledování a řazení do fronty – tuto funkci napojíme na přerušení, porty nebo systémová volání (podle možností operačního systému). Kdykoliv nastane událost, zjistíme potřebné údaje včetně vlastníka a zařadíme do fronty. Za určitých okolností tuto část můžeme nechat na zdrojovém jazyce, ve kterém píšeme překladač.
2. Událostní smyčka pro ošetření událostí – vybíráme události z fronty a spouštíme konkrétní funkce pro jejich ošetření. Událostní smyčka může vypadat takto:

   ```
   ZpracujZačátekProgramu;
   while not (ukončit_program) do
     if (je_něco_ve_frontě_událostí) then ZpracujUdálost(fronta_udalostí);
   ZpracujKonecProgramu;
   ```

Když programovací jazyk použitý pro vytvoření překladače neobsahuje podporu událostí, jednoduchá událostní smyčka s prioritami *konec* → *myš* → *klávesnice* je

```
ZpracujZačátekProgramu;
repeat
  if (ukončit_program) then begin
    ZpracujKonecProgramu;
    break;
  end;
  if (něco_je_na_portu_myši) then ZpracujMyš;
  if (něco_je_v_bufferu_klávesnice) then ZpracujKlávesnici;
until false;
```

6.5 Generování kódu v kompilátoru

Pro generování kódu můžeme použít atributovou gramatiku, která obsahuje sémantická pravidla vkládající do výstupního souboru instrukce assembleru.

Příklad 6.6

Vytvoříme atributovou gramatiku, která bude generovat assemblerový kód pro výpočet výrazů obsahujících operátory $+$ a $-$. Použijeme tyto instrukce assembleru:

Instrukce	*Význam*
`MOV dest, source`	přesune obsah druhého parametru do prvního, `dest = source`
`ADD dest, source`	přičte obsah druhého parametru k obsahu prvnímu, `dest = dest + source`
`SUB dest, source`	odečte obsah druhého parametru od prvního, `dest = dest - source`

Tabulka 6.3: Instrukce Assembleru pro sčítání a odčítání

Pro zjednodušení budeme předpokládat, že konstantní čísla i hodnoty proměnných jsou dvoubajtová čísla, a protože všechny tři instrukce obvykle vyžadují použití některého z datových registrů alespoň v jednom parametru, bude to vždy registr AX (zjednoduší to výpočet). Také adresaci velmi zjednodušíme.

Například pro řetězec `vysl = 25 + m + 3 - a` bude výstup

```
MOV AX, 25       ; znamená  AX = 25
ADD AX, m        ;          AX = AX + m
ADD AX, 3        ;          AX = AX + 3
SUB AX, a        ;          AX = AX - a
MOV vysl, AX     ;          vysl = AX
```

Syntaktická pravidla jsou tato (chceme $LL(1)$ gramatiku):

$S \to i = V$
$V \to AB$
$A \to n \mid i$
$B \to +V \mid -V \mid \varepsilon$

Sémantická pravidla „vmícháme“ dovnitř syntaktických, protože záleží také na čase, kdy se mají vykonat. Použijeme atributy $S[\,], V[dest, op], A[val], B[dest], n[lex], i[lex]$.

Atributy $dest$ a op jsou dědičné, do $dest$ ukládáme název proměnné, do které se má uložit výsledek, do op uložíme operátor, který se má provést. Zbylé atributy jsou syntetizované s obvyklým významem (pozor, do atributu val ukládáme řetězec, nikoliv číslo).

Atributová gramatika se sémantickými pravidly:

$S \to i = \ \{V.dest = \text{Promenna}(i.lex), V.op = \text{O\_NEDEF}\}\ V$
$V \to A\ \{\ B.dest = V.dest$
if $(V.op = \text{O\_PLUS})$ then Zapis$('\text{ADD}', '\text{AX}', A.val)$
else if $(V.op = O\_MINUS)$ then Zapis$('\text{SUB}', '\text{AX}', A.val)$
else Zapis$('\text{MOV}', '\text{AX}', A.val)\}\ B$
$A \to n\ \{A.val = n.lex\}$
$A \to i\ \{A.val = \text{Promenna}(i.lex)\}$
$B \to +\ \{V.dest = B.dest, V.op = \text{S\_PLUS}\}\ V$
$B \to -\ \{V.dest = B.dest, V.op = \text{S\_MINUS}\}\ V$
$B \to \varepsilon\ \{\text{Zapis}('\text{MOV}', B.dest, '\text{AX}')\}$

Sémantická funkce Zapis má tři parametry, které určují, co má být zapsáno jako instrukce kódu assembleru (jeden řádek) – první parametr je operátor, za ním následují dva operandy. Funkce zajistí správné formátování (například doplnění mezer a čárky mezi oba operandy).

Zatímco při interpretaci jsme řetězce překládali na čísla, zde je to naopak – také číselný operátor musí být lexikálním analyzátorem načten jako řetězec (pokud ovšem nechceme samotné číslo použít pro optimalizaci).

Assembler procesorů Intel vyžaduje v instrukcích uzavření názvů proměnných do hranatých závorek, tedy například použití proměnné X v instrukci je $[X]$. V našem příkladu provádí doplnění hranatých závorek funkce Promenna.

Pokud chceme vytvořit $LL(1)$ atributovou gramatiku překládající výrazy s více úrovněmi operátorů, je třeba použít dočasné proměnné pro uložení výsledků podvýrazů obsahujících pouze operátory vyšší priority. Ukázka takové atributové gramatiky je v příloze C na straně 203.

Úkoly ke kapitole 6

1. Postup interpretace výrazu dvěma zásobníky popsaný v kapitole 6.3 zjednodušte na pouze dvě úrovně priorit (jen aritmetické operátory) a naprogramujte včetně všech pomocných funkcí.
2. Zjistěte, zda je atributová gramatika z příkladu 6.6 na straně 175 opravdu $LL(1)$. Pokud ano, naprogramujte ji včetně všech pomocných funkcí.
3. Zamyslete se nad obdobnou atributovou gramatikou pro výrazy s násobením a dělením a také nad gramatikou obsahující operátory ve dvou úrovních priorit (první pro násobení a dělení, druhá pro sčítání a odčítání).

PŘÍLOHY

PŘÍLOHA A

Programovací jazyk popsaný $LL(1)$ atributovou gramatikou

A.1 Popis jazyka

Navrhneme interpretační překladač s touto lexikální a syntaktickou strukturou:

- matematické výrazy, proměnné (celočíselné), celá čísla,
- klíčová slova `begin`, `end`, `var`, `obdelnik`, `jestli`, `pak`, `cti`, `pis`,
- příkazy jsou *ukončeny* středníkem, možné příkazy jsou
  - **`var`** `prom1, prom2, ...;` – deklarace proměnných – příkaz `var` je vždy na začátku programu, ukončení středníkem je povinné, proměnné nemusí být žádné,
  - **`obdelnik`**`(x,y,s,v)` – vykreslí obdélník na souřadnice $[x, y]$ o šířce s a výšce v, parametry jsou výrazy,
  - **`cti`**`(prom)`,**`pis`**`(výraz)`,
  - **`begin ... end`** – složený příkaz, blok,
  - **`jestli`** `podmínka` **`pak`** `příkaz` – rozhodování, kde podmínka je ve tvaru `výraz < výraz` nebo `výraz = výraz`,
  - `prom := výraz` – přiřazovací příkaz, dvojznak `:=` se načte lexikálním analyzátorem jako jediný symbol.

Sémantické prvky jsou následující:

- proměnné jsou deklarovány na začátku programu, při deklaraci proměnnou zařadíme do tabulky symbolů,

- protože je umožněno používat příkaz typu `IF`, symboly mají dědičný atribut $prov$ (proveď), který je při nesplnění podmínky nastaven na $false$ a tato hodnota je také děděna u symbolů v složených příkazech (například u příkazu bloku),
- příkaz je proveden (interpretován) pouze tehdy, když má atribut $prov$ nastaven na $true$, jinak je pouze syntakticky rekurzívně rozvinut (pro kontrolu syntaktických chyb).

A.2 Popis struktury programu – gramatika

Nejdřív vytvoříme gramatiku typu $LL(1)$ popisující syntaktickou strukturu programu.

$S \to DbTe.$		$V \to AB$	výraz
$D \to vI$	deklarace	$B \to +AB$	
$I \to iJ$		$B \to -AB$	
$I \to ;$		$B \to \varepsilon$	
$J \to ,I$		$A \to CE$	
$T \to PR$	posloupnost příkazů	$E \to *CE$	
$T \to \varepsilon$		$E \to /CE$	
$R \to ;T$		$E \to \varepsilon$	
$P \to o(V,V,V,V)$	obdelnik	$C \to n$	
$P \to c(i)$	cti	$C \to i$	
$P \to p(V)$	pis	$C \to (V)$	
$P \to bTe$	blok příkazů	$M \to VN$	podmínka
$P \to jMtP$	jestli ...	$N \to = V$	
$P \to irV$	i := V	$N \to < V$	

Nyní je potřeba zkontrolovat, zda je tato gramatika $LL(1)$. Vytvoříme množiny $FOLLOW$:

$FOLLOW(S) = \{\$\}$	$FOLLOW(V) = \{',', ')', ';', =, <, t\}$
$FOLLOW(D) = \{b\}$	$FOLLOW(A) = \{+, -, ',', ')', ';', =, <, t\}$
$FOLLOW(I) = \{b\}$	$FOLLOW(B) = \{',', ')', ';', =, <, t\}$
$FOLLOW(J) = \{b\}$	$FOLLOW(C) = \{*, /, +, -, ',', ')', ';', =, <, t\}$
$FOLLOW(T) = \{e\}$	$FOLLOW(E) = \{+, -, ',', ')', ';', =, <, t\}$
$FOLLOW(R) = \{e\}$	$FOLLOW(M) = \{t\}$
$FOLLOW(P) = \{;\}$	$FOLLOW(N) = \{t\}$

S využitím dříve probíraných vzorců můžeme snadno zjistit, že tato gramatika je $LL(1)$.

K syntaktickým pravidlům přidáme sémantická pravidla a atributy a vytvoříme tak atributovou gramatiku.

$S \to Db\ \{T.prov = true\}\ Te.$

$D \to vI$

$I \to i\ \{\text{Pridej}(i.nazev)\}\ J$

$I \rightarrow ;$

$J \rightarrow , I$

$T \rightarrow \{P.prov = T.prov\}\ P\ \{R.prov = T.prov\}\ R$
$T \rightarrow \varepsilon$

$R \rightarrow ;\ \{T.prov = R.prov\}\ T$

$P \rightarrow \{V_0.prov = V_1.prov = V_2.prov = V_3.prov = P.prov\}\ o(V_0, V_1, V_2, V_3)$
$\{\text{if } P.prov \text{ then KresliObd}(V_0.val, V_1.val, V_2.val, V_3.val)\}$
$P \rightarrow c(i\ \{\text{if } P.prov \text{ then Zmen}(i.Nazev, \text{NactiZeVstupu})\}\ )$
$P \rightarrow p(\ \{V.prov = P.prov\}\ V\ \{\text{if } P.prov \text{ then Vypis}(V.val)\}\ )$
$P \rightarrow b\ \{T.prov = P.prov\}\ Te$
$P_0 \rightarrow j\ \{M.prov = P_0.prov\}\ Mt$
$\{\text{if } P_0.prov \text{ and } M.bool \text{ then } P_1.prov = true \text{ else } P_1.prov = false\}\ P_1$
$P \rightarrow ir\ \{V.prov = P.prov\}\ V\ \{\text{if } P.prov \text{ then Zmen}(i.nazev, V.val)\}$

$V \rightarrow \{A.prov = B.prov = V.prov\}$
$A\ \{\text{if } V.prov \text{ then } B.m = A.val\}\ B\ \{\text{if } V.prov \text{ then } V.val = B.val\}$
$B_0 \rightarrow \{A.prov = B.prov = V.prov\}$
$+\ A\ \{\text{if } B_0.prov \text{ then } B_1.m = B_0.m + A.val\}\ B_1\ \{\text{if } B_0.prov \text{ then } B_0.val = B_1.val\}$
$B_0 \rightarrow \{A.prov = B.prov = V.prov\}$
$-\ A\ \{\text{if } B_0.prov \text{ then } B_1.m = B_0.m - A.val\}\ B_1\ \{\text{if } B_0.prov \text{ then } B_0.val = B_1.val\}$
$B \rightarrow \varepsilon\ \{\text{if } B.prov \text{ then } B.val = B.m\}$

$A \rightarrow \{C.prov = E.prov = A.prov\}$
$C\ \{\text{if } A.prov \text{ then } E.m = C.val\}\ E\ \{\text{if } A.prov \text{ then } A.val = E.val\}$
$E_0 \rightarrow \{C.prov = E_1.prov = E_0.prov\}$
$*\ C\ \{\text{if } E_0.prov \text{ then } E_1.m = E_0.m * C.val\}\ E\ \{\text{if } E_0.prov \text{ then } E_0.val = E_1.val\}$
$E_0 \rightarrow \{C.prov = E_1.prov = E_0.prov\}$
$/\ C\ \{\text{if } E_0.prov \text{ then } E_1.m = E_0.m / C.val\}\ E_1\ \{\text{if } E_0.prov \text{ then } E_0.val = E_1.val\}$
$E \rightarrow \varepsilon\ \{\text{if } E.prov \text{ then } E.val = E.m\}$

$C \rightarrow n\ \{\text{if } C.prov \text{ then } C.val = n.lex\}$
$C \rightarrow i\ \{\text{if } C.prov \text{ then } C.val = \text{ DejHodnotu}(i.lex)\}$
$C \rightarrow (\ \{V.prov = C.prov\}\ V)\ \{\text{if } C.prov \text{ then } C.val = V.val\}$

$M \rightarrow \{V.prov = N.prov = M.prov\}$
$V\ \{\text{if } M.prov \text{ then } N.m = V.val\}\ N\ \{\text{if } M.prov \text{ then } M.bool = N.bool\}$
$N \rightarrow =\ \{V.prov = N.prov\}\ V\ \{\text{if } N.prov \text{ then } N.bool = (N.m == V.val)\}$
$N \rightarrow <\ \{V.prov = N.prov\}\ V\ \{\text{if } N.prov \text{ then } N.bool = (N.m < V.val)\}$

A.3 Práce se vstupem a lexikální analýza

Jsou definovány tyto datové typy a proměnné:

```
type
  TTypSymbolu = (S_ID, S_NUM, S_PLUS, S_MINUS, S_MUL, S_DIV, S_PRIRAD, S_LZAV,
    S_RZAV, S_STREDNIK, S_MENSI, S_ROVNO, S_OBDELNIK, S_CTI, S_PIS,
    S_JESTLI, S_PAK, S_VAR, S_BEGIN, S_END, S_ENDOFFILE, S_CARKA, S_TECKA);

  TNazevProm = string[15];

  TSymbol = record
    typ:        TTypSymbolu;
    atribcislo: integer;
    atribstr:   TNazevProm;
  end;

var
  symbol: TSymbol;
  znak:   TZnak;   // a další proměnné dle potřeby
```

Naprogramujeme proceduru `Lex`, která při každém zavolání načte jeden symbol ze vstupu. Typ symbolu uloží do globální proměnné `symbol.typ` a atribut symbolu do proměnné `symbol.atribcislo` v případě čísla nebo `symbol.atribstr` v případě řetězce (třeba název proměnné).

Zpracování symbolu rozdělíme na dvě části. V první části (přímé stavové programování) načteme symbol ze vstupu, druhou část (stav reprezentován proměnnou) použijeme jen v případě, že načtený řetězec začíná písmenem a mohlo by jít o klíčové slovo.

Nejdřív naprogramujeme druhou část, která slouží k rozpoznání klíčových slov. Sestavíme regulární gramatiku pro tento jazyk:

$L = \{begin, end, var, obdelnik, jestli, pak, cti, pis\}$

$S \rightarrow bA_1 \mid eA_5 \mid vA_7 \mid oA_9 \mid jA_{16} \mid pA_{21} \mid cA_{23}$

$A_1 \rightarrow eA_2$	$A_5 \rightarrow nA_6$	$S \rightarrow vA_7$	$A_9 \rightarrow bA_{10}$	$A_{16} \rightarrow eA_{17}$
$A_2 \rightarrow gA_3$	$A_6 \rightarrow d$	$A_7 \rightarrow aA_8$	$A_{10} \rightarrow dA_{11}$	$A_{17} \rightarrow sA_{18}$
$A_3 \rightarrow iA_4$		$A_8 \rightarrow r$	$A_{11} \rightarrow eA_{12}$	$A_{18} \rightarrow tA_{19}$
$A_4 \rightarrow n$			$A_{12} \rightarrow lA_{13}$	$A_{19} \rightarrow lA_{20}$
			$A_{13} \rightarrow nA_{14}$	$A_{20} \rightarrow i$
$A_{21} \rightarrow aA_{22} \mid iA_{25}$	$A_{23} \rightarrow tA_{24}$	$A_{25} \rightarrow s$	$A_{14} \rightarrow iA_{15}$	
$A_{22} \rightarrow k$	$A_{24} \rightarrow i$		$A_{15} \rightarrow k$	

Naprogramujeme proceduru `KlicoveSlovo`, která zjistí, zda je její parametr některým klíčovým slovem, a pokud ano, změní podle svého výsledku typ načteného symbolu. Nejdřív stanovíme konstanty pro chybové stavy a koncový stav.

```
procedure KlicoveSlovo(slovo: string);
const
  k_chyba = 26;   k_end = 28;   k_obdelnik = 30;   k_pak = 32;   k_pis = 34;
  k_begin = 27;   k_var = 29;   k_jestli   = 31;   k_cti = 33;
var
  stav: byte;
  znak: char;
  delka, pozice: byte;
begin
  stav := 0;
  delka := length(slovo);
  pozice := 1;
  while (pozice <= delka) and (stav < k_chyba) do begin
    znak := slovo[pozice];
    case stav of
      0: begin case znak of
           'B': stav := 1;  'V': stav := 7;  'J': stav := 16;  'C': stav := 23;
           'E': stav := 5;  'O': stav := 9;  'P': stav := 21;
           else stav := k_chyba;
         end;
      1: if znak = 'E' then stav := 2 else stav := k_chyba;
      2: if znak = 'G' then stav := 3 else stav := k_chyba;
      3: if znak = 'I' then stav := 4 else stav := k_chyba;
      4: if znak = 'N' then stav := k_begin else stav := k_chyba;
      ...  // atd. pro všechny stavy až do 25
    end; // case
    inc(pozice);
  end; // while
  if pozice < delka then stav := k_chyba;  // případ, kdy slovo jen začíná jako
  case stav of                             // klíčové, například BEGINNER
    k_begin: symbol.typ := S_BEGIN;
    k_end:   symbol.typ := S_END;
    ...  // atd. pro všechna klíčová slova
    k_chyba: symbol.typ := S_ID;
  end    // case
end      // procedure
```

Předpokládáme, že je již naprogramována funkce `DejZnak`, a to naprosto stejně jako na straně 24. Funkce `Lex` zpracovává vstup metodou přímého stavového programování, jsou ošetřeny lexikální chyby.

```
procedure Lex;
begin
  while (znak.rad[znak.pozice] = ' ') do Dejznak;
  case znak.rad[znak.pozice] of
    'A'..'Z': begin              // identifikátor nebo klíčové slovo, délka > 0
      symbol.atribstr := znak.rad[znak.pozice];
      DejZnak;
      while (znak.rad[znak.pozice] in ['A'..'Z', '0'..'9']) do begin
        symbol.atribstr := symbol.atribstr + znak.rad[znak.pozice];
        DejZnak;
      end;
```

```
      KlicoveSlovo(symbol.atribstr);// také načte správnou hodnotu do symbol.typ
    end;
    '0'..'9': begin                 // číslo
      symbol.atribcislo := znak.rad[znak.pozice];
      DejZnak;
      while (znak.rad[znak.pozice] in ['0'..'9']) do begin
        symbol.atribcislo := symbol.atribcislo + znak.rad[znak.pozice];
        DejZnak;
      end;
      if (znak.rad[znak.pozice] in ['A'..'Z']) then
        error('za číslicí nemůže být písmeno');
      symbol.typ := S_NUM;
    end;
    '+': begin
      symbol.typ := S_PLUS;
      DejZnak;
    end;
    ... // atd. pro všechny typy symbolů
    else error('neznámý symbol');
  end;
end;
```

A.4 Implementace tabulky symbolů

Možnosti implementace jsou různé, jedna z nejjednodušších je třeba spojový seznam. Je podobná implementaci navržené v příkladu 4.3 na straně 100, ale mnohem jednodušší a kompaktnější, protože nepoužíváme uživatelsky definované datové typy ani funkce a proměnné jsou pouze celočíselné.

```
type
  PPolozkaTab = ↑TPolozkaTab;
  TPolozkaTab = record
    nazev: TNazevProm;
    hodnota: integer;
    dalsi: PPolozkaTab;
  end;
var Tabulka: PPolozkaTab;  // ukazatel na první prvek tabulky

procedure InicializujTabulku(var tab: PPolozkaTab);
procedure ZnicTabulku(var tab: PPolozkaTab);

procedure Najdi(var ukaz_tab: PPolozkaTab; nazev: TNazev);
// v ukazateli ukaz_tab vrátí ukazatel na tu položku tabulky, ve které
// je proměnná s daným názvem, když v seznamu není, vrátí nil

procedure Pridej(var tab: PPolozkaTab; nazev: TNazev; hodnota: integer);
// jestliže je tabulka prázdná, vytvoří první položku s těmito údaji,
// jestliže ne, najde dané místo v tabulce a přidá záznam
// kdyby už položka existovala, hlásí sémantickou chybu
```

```
procedure Zmen(var tab: PPolozkaTab; nazev: TNazev; hodnota: integer);
// ověří, zda položka s tímto názvem existuje, když ne, hlásí
// sémantickou chybu, když ano, provede změnu hodnoty

function DejHodnotu(tab: PPolozkaTab; nazev: TNazev): integer;
// když položka s tímto názvem neexistuje, hlásí sémantickou chybu
```

První parametr těchto funkcí můžeme vynechat, pokud budeme mít jen jedinou (globální) tabulku. Jestliže chceme použít blokovou strukturu a odlišovat různé úrovně lokálních proměnných, navíc přidáme dynamický zásobník, do kterého budeme řadit tabulky bloků, a vyhledávací proceduru upravíme tak, aby začala vyhledávat na vrcholu zásobníku a pokračovala v něm dále směrem dolů.

Jinou podobnou implementací je binární strom, ve kterém by se navíc zjednodušilo a zrychlilo vyhledávání. Rozdíl je jenom v uspořádání položek – do datového typu `TPolozkaTab` dáme místo odkazu na následníka dva odkazy (levý a pravý potomek).

A.5 Překlad rekurzivním sestupem

Vytvoříme funkci, která bude volána kdykoliv, když v pravidle narazíme na vstupní terminální symbol (výstupní terminály nepoužíváme, neřešíme). Tato funkce zajistí veškerou práci se vstupem, tj. porovnání terminálu v pravidle s příslušným symbolem na vstupu, a následné zavolání funkce `Lex`, která taktéž provede posun na další symbol na vstupu.

```
procedure expect(typ: TTypSymbolu);
begin
  if typ = sym.typ then Lex
    else error('Symbol '+VypisHodn(sym)+' není očekávaného typu '+VypisTyp(typ));
end;
```

Tato procedura je použitelná pro obě metody implementace popsané v předchozích kapitolách – přepis rozkladové tabulky i rekurzivní sestup (pro přepis rozkladové tabulky je v popisu metod nazvána `pop`, ale vnitřně je stejná). Další funkce (procedury):

- `S,D,I,J,T(prov: boolean),R(prov: boolean),P(prov: boolean),`
- `V(prov: boolean; var val: integer),`
  `A(prov: boolean; var val: integer),`
- `B(prov: boolean; m: integer; var val: integer),`
  `E(prov: boolean; m: integer; var val: integer),`
- `C(prov: boolean; var val: integer),`
- `M(prov: boolean; var bool: integer),`
- `N(prov: boolean; m: integer; var bool: integer),`
- `Pridej, Zmen, DejHodnotu` – pro práci s tabulkou,
- `KresliObd` – pro vykreslení obdélníka,
- `NactiZeVstupu, Vypis` – pro práci se vstupem a výstupem.

Atribut *prov* a jeho testování lze vynechat, pokud nepoužijeme rozhodovací příkaz. Naprogramujeme všechny procedury rekurzivního sestupu. Procedura `error` pro ošetření chyby má stejnou syntaxi, kterou známe z předchozích kapitol, tedy vypíše pozici ve zdrojovém kódu a následně řetězec s chybovým hlášením (použili jsme ji už v proceduře `Lex`). Obdobně pracují funkce vracející řetězcovou reprezentaci datového typu nebo symbolu.

```
procedure S;
begin
  if symbol.typ = S_BEGIN then begin
    D;
    expect(S_BEGIN);
    T(true);
    expect(S_END);
    expect(S_TECKA);
  end else error('místo symbolu '+VypisHodn(sym)+' očekáván '+VypisTyp(S_BEGIN));
end;

procedure D;
begin
  if symbol.typ = S_VAR then begin
    expect(S_VAR);
    I;
  end else error('místo symbolu '+VypisHodn(sym)+' očekáván '+VypisTyp(S_VAR));
end;

procedure I;
begin
  case symbol.typ of
    S_ID: begin
      Pridej(symbol.atribstr);
      expect(S_ID);
      J;
    end;
    S_STREDNIK: expect(S_STREDNIK);
    else error('místo symbolu '+VypisHodn(sym)+' očekáván '
      +VypisTyp(S_ID)+' nebo '+VypisTyp(S_STREDNIK));
  end;
end;

procedure J;
begin
  if symbol.typ = S_CARKA then begin
    expect(S_CARKA);
    I;
  end else error('místo symbolu '+VypisHodn(sym)+' očekáván '+VypisTyp(S_CARKA));
end;

procedure T(prov: boolean);
begin
  case symbol.typ of
    S_OBDELNIK,S_CTI,S_PIS,S_JESTLI,S_ID: begin
```

```
      P(prov);
      R(prov);
    end;
    S_END: ;
    else error(...);   // chyba ošetřena podobně jako v předchozích procedurách,
  end;                 // totéž platí i dále
end;

procedure R(prov: boolean);
begin
  if symbol.typ = S_STREDNIK then begin
    expect(S_STREDNIK);
    T(prov);
  end else error(...);
end;

procedure V(prov: boolean; var val: integer);
var m: integer;
begin
  if symbol.typ in [S_NUM,S_ID,S_LZAV] then begin
    A(prov,m);
    B(prov,m,val);
  end else error(...);
end;

procedure B(prov: boolean; m: integer; var val: integer);
var pomval: integer;
begin
  case symbol.typ of
    S_PLUS: begin
      expect(S_PLUS);
      A(prov,pomval);
      B(prov,m+pomval,val);
    end;
    S_MINUS: begin
      expect(S_MINUS);
      A(prov,pomval);
      B(prov,m-pomval,val);
    end;
    S_CARKA,S_RZAV,S_STREDNIK,S_MENSI,S_ROVNO,S_PAK:
      if prov then val := m;
    else error(...);
  end;
end;

procedure A(prov: boolean; var val: integer);
var m: integer;
begin
  if symbol.typ in [S_NUM,S_ID,S_LZAV] then begin
    C(prov,m);
    E(prov,m,val);
  end else error(...);
end;
```

```
procedure E(prov: boolean; m: integer; var val: integer);
var pomval: integer;
begin
  case symbol.typ of
    S_MUL: begin
      expect(S_MUL);
      C(prov,pomval);
      E(prov,m*pomval,val);
    end;
    S_DIV: begin
      expect(S_DIV);
      C(prov,pomval);
      if pomval = 0 then error('dělení nulou') else E(prov,m/pomval,val);
    end;
    S_PLUS,S_MINUS,S_CARKA,S_RZAV,S_STREDNIK,S_MENSI,S_ROVNO,S_PAK:
      if prov then val := m;
    else error(...);
  end;
end;

procedure C(prov: boolean; var val: integer);
begin
  case symbol.typ of
    S_NUM: begin
      val := symbol.atribcislo;
      expect(S_NUM);
    end;
    S_ID: begin
      val := ZjistiHodnotu(symbol.atribstr);
      expect(S_ID);
    end;
    S_LZAV: begin
      expect(S_LZAV);
      A(prov,val);
      expect(S_RZAV);
    end;
    else error(...);
  end;
end;

procedure M(prov: boolean; var bool: integer);
var m: integer;
begin
  if symbol.typ in [S_NUM,S_ID,S_LZAV] then begin
    V(prov,m);
    N(prov,m,bool);
  end else error(...);
end;

procedure N(prov: boolean; m: integer; var bool: integer);
var pomm: integer;
begin
```

```
  case symbol.typ of
    S_ROVNO: begin
      expect(S_ROVNO);
      V(prov,pomm);
      bool := (m = pomm);
    end;
    S_MENSI: begin
      expect(S_MENSI);
      V(prov,pomm);
      bool := (m < pomm);
    end;
    else error(...);
  end;
end;
```

Nejkomplikovanější je procedura `P` generující příkazy. Nejdřív sestavíme základní tvar procedury. Následuje vnitřní část pro jednotlivá pravidla odpovídající příkazům zdrojového jazyka. Vždy za pravidlem se sémantikou je větev příkazu `case` pro dané pravidlo.

```
procedure P(prov: boolean);
var v1, v2, v3, v4: integer; b: boolean; s: NazevProm;
begin
  case sym.typ of
  ...                  // jednotlivé hodnoty typů symbolů
  else error(...);     // chyba ošetřena podobně jako v předchozích procedurách
  end;   // case
end;
```

$P \rightarrow \{V_0.prov = V_1.prov = V_2.prov = V_3.prov = P.prov\}\ o(V_0, V_1, V_2, V_3)$
$\quad \{\text{if } P.prov \text{ then KresliObd}(V_0.val, V_1.val, V_2.val, V_3.val)\}$

```
S_OBDELNIK: begin
  expect(S_OBDELNIK);
  expect(S_LZAV);
  V(prov,v1);  expect(S_CARKA);
  V(prov,v2);  expect(S_CARKA);
  V(prov,v3);  expect(S_CARKA);
  V(prov,v4);  expect(S_RZAV);
  if prov then KresliObd(v1,v2,v3,v4);
end;
```

$P \rightarrow c(i\ \{\text{if } P.prov \text{ then Zmen}(i.Nazev, \text{NactiZeVstupu})\}\ )$

```
S_CTI: begin
  expect(S_CTI); expect(S_LZAV);
  if sym.typ = S_ID then s := sym.atribstr;
  expect(S_ID);
  if prov then begin
```

```
    NactiZeVstupu(v1);
    Zmen(s,v1);
  end;
  expect(S_RZAV);
end;
```

$P \rightarrow p(\ \{V.prov = P.prov\}\ V\ \{\text{if } P.prov \text{ then Vypis}(V.val)\}\ )$

```
S_PIS: begin
  expect(S_PIS); expect(S_LZAV);
  V(prov,v1);
  if prov then Vypis(v1);
  expect(S_RZAV);
end;
```

$P \rightarrow b\ \{T.prov = P.prov\}\ Te$

```
S_BEGIN: begin
  expect(S_BEGIN);
  T(prov);
  expect(S_END);
end;
```

$P_0 \rightarrow j\ \{M.prov = P_0.prov\}\ Mt$
$\{\text{if } P_1.prov \text{ and } M.bool \text{ then } P_1.prov = true \text{ else } P_1.prov = false\}\ P_1$

```
S_JESTLI: begin
  expect(S_JESTLI);
  M(prov,b);
  expect(S_PAK);
  P(prov and b);
end;
```

$P \rightarrow ir\ \{V.prov = P.prov\}\ V\ \{\text{if } P.prov \text{ then Zmen}(i.nazev, V.val)\}$

```
S_ID: begin
  s := sym.atribstr;
  expect(S_ID);
  expect(S_PRIRAD);
  V(prov,v1);
  if prov then Zmen(s,v1);
end;
```

Zbývá napsat hlavní funkci celé analýzy:

```
procedure Analyza;
begin
  init;       // inicializace překladu včetně lexikálního analyzátoru
  Lex;        // přednačteme jeden symbol
  S;          // spustíme rekurzivní volání
  done;       // ukončení překladu, úklid paměti apod.
end;
```

PŘÍLOHA B

Silná $LR(1)$ atributová gramatika programovacího jazyka

B.1 Popis jazyka

Naprogramování LR překladu popíšeme na jiném jazyce. Jde o jednoduchý jazyk na ovládání postavičky (robota, želvičky apod.). Lexikální a syntaktická struktura je následující:

- matematické výrazy, proměnné (celočíselné), celá čísla,
- klíčová slova `beginprog`, `endprog`, `beginvar`, `endvar`, `left`, `right`, `go`, `message`, `wait`,
- možné příkazy:
  - **`beginvar`** `prom1, prom2, ...` **`endvar`** – deklarace proměnných – alespoň jedna proměnná musí být deklarována, příkaz musí být na začátku vstupu,
  - **`beginprog`** `...` **`endprog`** – vymezení začátku a konce programu, následuje po deklaracích,
  - **`go`**`[výraz]` – postavička popojde o tolik kroků, na kolik je vyhodnocen výraz,
  - **`left`**, **`right`** – otočení postavičky o $90\,\mathrm{deg}$ doleva nebo doprava,
  - **`message`**`[výraz]` – vypíše hodnotu výrazu jako zprávu do zvláštního okénka,
  - **`wait`**`[výraz]` – provádění programu se zastaví na tolik časových úseků, kolik je stanoveno výrazem,
  - `prom = vyraz` – přiřazovací příkaz,
- příkazy jsou *navzájem odděleny* středníkem.

Sémantické prvky jsou následující:

- každá použitá proměnná musí být deklarována, při deklaraci je proměnná zařazena do tabulky symbolů a inicializována na 0,
- hrací plocha je vymezena pevnými hranicemi,
- při pohybu postavičky je třeba kontrolovat, zda není u okraje hrací plochy, tento okraj nesmí překročit (příkaz pro pohyb není vykonán),
- existují sémantické funkce zajišťující grafiku programu, pohyb postavičky a jeho kontrolu, výpis výsledku výrazu do zvláštního (modálního) okna, čekání na daný počet sekund, a také pro práci s tabulkou symbolů.

B.2 Popis struktury programu – gramatika

Syntaktická pravidla jsou následující (uvádíme rozšířenou gramatiku):

$S' \to \#S$	⓪	$FOLLOW(S') = \{\$\}$
$S \to D\ bp\ T\ ep$	①	$FOLLOW(S) = \{\$\}$
$D \to bv\ I$	②	$FOLLOW(D) = \{bp\}$
$I \to Ii, \mid i\ ev$	③,④	$FOLLOW(I) = \{i, bp\}$
$T \to RP$	⑤	$FOLLOW(T) = \{ep, ;\}$
$R \to T; \mid \varepsilon$	⑥,⑦	$FOLLOW(R) = \{l, r, g, m, w, i\}$
$P \to l \mid r \mid Z[V] \mid i = V$	⑧,⑨,⑩,⑪	$FOLLOW(P) = \{ep, ;\}$
$Z \to g \mid m \mid w$	⑫,⑬,⑭	$FOLLOW(Z) = \{[\}$
$V \to AM$	⑮	$FOLLOW(V) = \{+, -,), ep, ;\}$
$A \to V+ \mid V- \mid \varepsilon$	⑯,⑰,⑱	$FOLLOW(A) = \{n, i, (\}$
$M \to BF$	⑲	$FOLLOW(M) = \{*, /, +, -,), ep, ;\}$
$B \to M* \mid M/ \mid \varepsilon$	⑳,㉑,㉒	$FOLLOW(B) = \{n, i, (\}$
$F \to n \mid i \mid (V)$	㉓,㉔,㉕	$FOLLOW(F) = \{*, /, +, -,), ep, ;\}$

Zjistíme, zda je tato gramatika silná $LR(1)$. K tomu potřebujeme kromě množin $FOLLOW$ také množiny $BEFORE$ a některé množiny EFF.

$BEFORE(S') = \{\#\}$
$BEFORE(S) = \{\#\}$
$BEFORE(D) = \{\#\}$
$BEFORE(I) = \{bv\}$
$BEFORE(T) = \{bp\}$
$BEFORE(R) = \{bp\}$
$BEFORE(P) = \{R\}$
$BEFORE(Z) = \{R\}$
$BEFORE(V) = \{[, =, (\}$
$BEFORE(A) = \{[, =, (\}$
$BEFORE(M) = \{A\}$
$BEFORE(B) = \{A\}$
$BEFORE(F) = \{B\}$

Otestujeme, zda je gramatika silná $LR(1)$. Budeme postupovat podle bodů určených v definici 3.21 silné $LR(k)$ gramatiky na straně 82.

1. (a) Žádná dvě pravidla nekončí stejným symbolem, není co testovat.
   (b) Množiny $BEFORE$ žádného ze tří ε-pravidel neobsahují symbol, který by byl posledním symbolem řetězců na pravých stranách pravidel, tedy není co testovat.
   (c) Množiny $BEFORE$ neterminálů s ε-pravidly mají po dvou vždy prázdné průniky, proto není co testovat.

2. (a)
$I \to Ii,\ D \to \textbf{b}\,\textbf{v}\,I$ $\quad EFF(i, \cdot FOLLOW(I)) \cap FOLLOW(D) = \emptyset$
$I \to Ii,\ F \to i$ $\quad EFF(, \cdot FOLLOW(I)) \cap FOLLOW(F) = \emptyset$
$I \to i\,\textbf{e}\,\textbf{v},\ F \to i$ $\quad EFF(\textbf{e}\,\textbf{v} \cdot FOLLOW(I)) \cap FOLLOW(F) = \emptyset$
$P \to Z[V],\ P \to i = V$ $\quad EFF(\,] \cdot FOLLOW(P)) \cap FOLLOW(P) = \emptyset$
$P \to i = V,\ F \to i$ $\quad EFF(= V \cdot FOLLOW(P)) \cap FOLLOW(F) = \emptyset$
$A \to V+,\ P \to i = V$ $\quad EFF(+ \cdot FOLLOW(A)) \cap FOLLOW(P) = \emptyset$
$A \to V-,\ P \to i = V$ $\quad EFF(- \cdot FOLLOW(A)) \cap FOLLOW(P) = \emptyset$
$B \to M*,\ V \to AM$ $\quad EFF(* \cdot FOLLOW(B)) \cap FOLLOW(V) = \emptyset$
$B \to M/,\ V \to AM$ $\quad EFF(/ \cdot FOLLOW(B)) \cap FOLLOW(V) = \emptyset$
$F \to (V),\ P \to i = V$ $\quad EFF(\,) \cdot FOLLOW(F)) \cap FOLLOW(P) = \emptyset$
   (b)
$R \to \varepsilon,\ S \to D\,\textbf{b}\,\textbf{p}\,T\,\textbf{e}\,\textbf{p}$ $\quad EFF(T\,\textbf{e}\,\textbf{p} \cdot FOLLOW(S)) \cdot FOLLOW(R) = \emptyset$
$A \to \varepsilon,\ P \to Z[V]$ $\quad EFF(V] \cdot FOLLOW(P)) \cap FOLLOW(A) = \emptyset$
$A \to \varepsilon,\ P \to i = V$ $\quad EFF(V \cdot FOLLOW(P)) \cap FOLLOW(A) = \emptyset$
$A \to \varepsilon,\ F \to (V)$ $\quad EFF(V) \cdot FOLLOW(F)) \cap FOLLOW(A) = \emptyset$
$B \to \varepsilon,\ V \to AM$ $\quad EFF(M \cdot FOLLOW(V)) \cap FOLLOW(B) = \emptyset$
   (c)
$R \to \varepsilon,\ T \to RP$ $\quad EFF(RP \cdot FOLLOW(T)) \cap FOLLOW(R) = \emptyset$
$A \to \varepsilon,\ V \to AM$ $\quad EFF(AM \cdot FOLLOW(V)) \cap FOLLOW(A) = \emptyset$
$B \to \varepsilon,\ M \to BF$ $\quad EFF(BF \cdot FOLLOW(M)) \cap FOLLOW(B) = \emptyset$

Jde o silnou $LR(1)$ gramatiku, můžeme vytvořit rozkladovou tabulku, kterou pak použijeme pro naprogramování syntaxe metodou přepisu rozkladové tabulky. Ta je velmi rozsáhlá, zabírá následující dvě strany.

Vytvoříme atributovou gramatiku.

$S' \to \#S$ ⓪
$S \to D\,\textbf{b}\,\textbf{p}\,T\,\textbf{e}\,\textbf{p}$ ①
$D \to \textbf{b}\,\textbf{v}\,I$ ②
$I \to Ii,\ \{\text{Pridej}(i.nazev)\}$ ③
$I \to i\,\textbf{e}\,\textbf{v}\ \{\text{Pridej}(i.nazev)\}$ ④
$T \to RP$ ⑤
$R \to T;$ ⑥
$R \to \varepsilon$ ⑦
$P \to l\ \{turn\_left\}$ ⑧

	n	i	$+$	$-$	$*$	$/$	$($	$)$	$[$	$]$	$,$	$;$	$\ldots$
S'													
S													
D													
I		*push*											
T												*push*	
R		*push*											
P												r5	
Z									*push*				
V								*push*		*push*		r11	
A	r22	r22					r22						
M			r15	r15	*push*	*push*		r15				r15	
B	*push*	*push*					*push*						
F			r19	r19	r19	r19		r19				r19	
n			r23	r23	r23	r23		r23				r23	
i			r24	r24	r24	r24		r24			*push*	r24	
$+$	r16	r16					r16						
$-$	r17	r17					r17						
$*$	r20	r20					r20						
$/$	r21	r21					r21						
$($	r18	r18					r18						
$)$			r25	r25	r25	r25		r25				r25	
$[$	r18	r18					r18					r10	
$]$													
$,$		r3											
$;$		r6											
bp		r7											
ep													
bv		*push*											
ev		r4											
$=$	r18	r18					r18						
l												r8	
r												r9	
g									r12				
m									r13				
w									r14				
$\#$													

$P \to r \ \{turn\_right\}$ ⑨

$P \to Z[V] \ \Big\{\text{case } Z.op \text{ of}\Big(\text{go}(V.val), \text{message}(V.val), \text{wait}(V.val)\Big)\Big\}$ ⑩

$P \to i = V \ \{\text{Zmen}(i.nazev, V.val)\}$ ⑪

$Z \to g \ \{Z.op = \text{S\_GO}\}$ ⑫

$Z \to m \ \{Z.op = \text{S\_MESSAGE}\}$ ⑬

$Z \to w \ \{Z.op = \text{S\_WAIT}\}$ ⑭

$V \to AM \ \{\text{if } A.op = \text{S\_PLUS then } V.val = A.val + M.val$ ⑮
$\text{else } V.val = A.val - M.val\}$

$A \to V + \ \{A.val = V.val, A.op = \text{S\_PLUS}\}$ ⑯

	...	*bp*	*ep*	*bv*	*ev*	=	*l*	*r*	*g*	*m*	*w*	$
S'												*acc*
S												r0
D		*push*										
I		r2										
T			*push*									
R							*push*	*push*	*push*	*push*	*push*	
P			r5									
Z												
V			r11									
A												
M			r15									
B												
F			r19									
n			r23									
i			r24		*push*	*push*						
$+$												
$-$												
$*$												
$/$												
(												
)			r25									
[												
]		r10										
,		r3										
;							r6	r6	r6	r6	r6	
bp							r7	r7	r7	r7	r7	
ep												r1
bv												
ev		r4										
=												
l			r8									
r			r9									
g												
m												
w												
#				*push*								

$A \to V -\ \ \{A.val = V.val, A.op = \text{S\_MINUS}\}$ ⑰

$A \to \varepsilon\ \ \{A.val = 0, A.op = \text{S\_PLUS}\}$ ⑱

$M \to BF\ \ \{\text{if } B.op = \text{S\_MUL then } M.val = B.val * F.val$ ⑲
$\text{else } M.val = B.val/F.val\}$

$B \to M *\ \ \{B.val = M.val, B.op = \text{S\_MUL}\}$ ⑳

$B \to M/\ \ \{B.val = M.val, B.op = \text{S\_DIV}\}$ ㉑

$B \to \varepsilon\ \ \{B.val = 1, B.op = \text{S\_MUL}\}$ ㉒

$F \to n\ \ \{F.val = n.lex\}$ ㉓

$F \to i\ \ \{F.val = \text{ZjistiHodnotu}(i.nazev)\}$ ㉔

$F \to (V)\ \ \{F.val = V.val\}$ ㉕

B.3 Implementace řízení překladu

Deklarujeme potřebné datové typy a proměnné.

```
type
  TTypSymbolu = (S_ID, S_NUM, S_ROVNASE, S_PLUS, S_MINUS,       // terminály
    S_MUL, S_DIV, S_LPAR, S_RPAR, S_LHPAR, S_RHPAR, S_CARKA,
    S_ENDOFFILE, S_BEGVAR, S_ENDVAR, S_BEGPROG, S_ENDPROG,
    S_LEFT, S_RIGHT, S_GO, S_MESSAGE, S_WAIT, S_STREDNIK,
    S_NSC, S_NS, S_ND, S_NI, S_NT, S_NR, S_NP, S_NZ,           // neterminály
    S_NV, S_NA, S_NM, S_NB, S_NF, S_HASH);

  TSymbol = record
    typ:        TTypSymbolu;
    atribcislo: integer;
    atribstr:   string;
  end;

  TSymbolZasob = record
    typ:        TTypSymbolu;
    atribcislo: integer;
    atribstr:   string;
    atribop:    TTypSymbolu; // S_PLUS, S_MINUS, S_MUL, S_DIV, S_GO, ...
  end;

var
  konec:      boolean;        // indikátor ukončení výpočtu, proveden accept
  symbol:     TSymbol;        // aktuální symbol načtený z proměnné vstup
  vrchol_zas: TSymbolZasob;   // symbol na vrcholu zásobníku
  zasobnik:   TZasobnik;      // prvky jsou typu TSymbolZasob
  ...         // další potřebné datové typy a proměnné
```

Proceduru `Lex` necháváme na čtenáři, programuje se podobně jako v příloze A. Definujeme inicializaci a ukončení, a dále řídicí procedury tabulky a celé analýzy:

```
procedure Init;
var SymbolZas: TSymbolZas;
begin
  ...                                 // inicializace vstupu a výstupu
  Vytvor_zasobnik;
  SymbolZas.typ := S_HASH;
  Pridej_do_zasobniku(SymbolZas);   // nelze napsat přímo S_HASH!
  Lex;
  Konec := false;
end;

procedure Done;
begin
  Zlikviduj_zasobnik;          // uvolní paměť zabranou zásobníkem
  ...                          // uzavření vstupu a výstupu
end;
```

```
procedure Akce;
begin
  case vrchol_zas of
    S_NSC: if symbol.typ = S_ENDOFFILE then accept
      else error('očekáván konec zdrojového souboru');
    S_NS: if symbol.typ = S_ENDOFFILE then reduce(0)
       else error('očekáván konec zdrojového souboru');
    S_ND: if symbol.typ = S_BEGPROG then push
       else error('očekáván začátek programu');
    S_NI: case symbol.typ of
        S_ID: push;
        S_BEGPROG: reduce(2);
        else error('místo symbolu '+VypisHodn(symbol)+' očekávána další '
          'proměnná nebo začátek programu');
      end;
    S_NT: if symbol.typ in [S_STREDNIK,S_ENDPROG] then push
      else error(...);  // tato i následující chyby jsou ošetřeny podobně
                        // jako předchozí
    S_NR: if symbol.typ in [S_ID,S_LEFT,S_RIGHT,S_GO,S_MESSAGE,S_WAIT] then push
      else error(...);
    S_NP: if symbol.typ in[S_STREDNIK,S_ENDPROG] then reduce(5)
      else error(...);
    S_NZ: if symbol.typ = S_LHPAR then push
      else error(...);
    S_NV: case symbol.typ of
        S_RPAR,S_RHPAR: push;
        S_STREDNIK,S_ENDPROG: reduce(11);
        else error(...);
      end;
    S_NA: if symbol.typ in [S_NUM,S_ID,S_LPAR] then reduce(22)
      else error(...);
    S_NM: case symbol.typ of
        S_MUL,S_DIV: push;
        S_PLUS,S_MINUS,S_RPAR,S_STREDNIK,S_ENDPROG: reduce(15);
        else error(...);
      end;
    S_NB: if symbol.typ in [S_NUM,S_ID,S_LPAR] then push
      else error(...);
    S_NF: if symbol.typ in [S_PLUS,S_MINUS,S_MUL,S_DIV,S_RPAR,S_STREDNIK,
      S_ENDPROG] then reduce(19)
      else error(...);
    S_NUM: if symbol.typ in [S_PLUS,S_MINUS,S_MUL,S_DIV,S_RPAR,S_STREDNIK,
      S_ENDPROG] then reduce(23)
      else error(...);
    S_PLUS: if symbol.typ in [S_NUM,S_ID,S_LPAR] then reduce(16)
      else error(...);
    S_MINUS: if symbol.typ in [S_NUM,S_ID,S_LPAR] then reduce(17)
      else error(...);
    S_MUL: if symbol.typ in [S_NUM,S_ID,S_LPAR] then reduce(20)
      else error(...);
```

```
    S_DIV: if symbol.typ in [S_NUM,S_ID,S_LPAR] then reduce(21)
      else error(...);
    S_LPAR: if symbol.typ in [S_NUM,S_ID,S_LPAR] then reduce(18)
      else error(...);

    ...    // atd. pro terminály S_RPAR, S_LHPAR, S_RHPAR, S_CARKA,
           // S_STREDNIK, S_BEGPROG, S_ENDPROG, S_BEGVAR, S_ENDVAR,
           // S_ROVNASE, S_LEFT, S_RIGHT, S_GO, S_MESSAGE, S_WAIT

    S_HASH: if symbol.typ = S_BEGVAR then push
      else error(...);
    else error(...);
  end;  // case
end;

procedure Analyza;
begin
  Init;
  while (not Konec) do Akce;
  Done;
end;
```

B.4 Implementace operací v tabulce

Podle pravidel atributové gramatiky napíšeme proceduru pro redukci. Proceduru větvíme podle čísla použitého pravidla gramatiky.

```
procedure reduce(cislo_prav: integer);
var
  SymbolZas: TSymbolZas;
  val: integer;
begin
  case cislo_prav of
    0: begin
      Vyjmi_ze_zasobniku(vrchol_zas);              // S
      Vyjmi_ze_zasobniku(vrchol_zas);              // #
      SymbolZas.typ := S_NSC;
      Pridej_do_zasobniku(SymbolZas);
    end;
    1: begin
      Vyjmi_ze_zasobniku(vrchol_zas);              // ep
      Vyjmi_ze_zasobniku(vrchol_zas);              // T
      Vyjmi_ze_zasobniku(vrchol_zas);              // bp
      Vyjmi_ze_zasobniku(vrchol_zas);              // D
      SymbolZas.typ := S_NS;
      Pridej_do_zasobniku(SymbolZas);
    end;
    2: begin
      Vyjmi_ze_zasobniku(vrchol_zas);              // I
      Vyjmi_ze_zasobniku(vrchol_zas);              // bv
```

```
  SymbolZas.typ := S_ND;
  Pridej_do_zasobniku(SymbolZas);
end;
3: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // ,
  Vyjmi_ze_zasobniku(vrchol_zas);              // i
  Pridej(vrchol_zas.atribstr);
  Vyjmi_ze_zasobniku(vrchol_zas);              // I
  SymbolZas.typ := S_NI;
  Pridej_do_zasobniku(SymbolZas);
end;
4: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // ev
  Vyjmi_ze_zasobniku(vrchol_zas);              // i
  Pridej(vrchol_zas.atribstr);
  SymbolZas.typ := S_NI;
  Pridej_do_zasobniku(SymbolZas);
end;
5: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // P
  Vyjmi_ze_zasobniku(vrchol_zas);              // R
  SymbolZas.typ := S_NT;
  Pridej_do_zasobniku(SymbolZas);
end;
6: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // ;
  Vyjmi_ze_zasobniku(vrchol_zas);              // T
  SymbolZas.typ := S_NR;
  Pridej_do_zasobniku(SymbolZas);
end;
7: begin
  SymbolZas.typ := S_NR;
  Pridej_do_zasobniku(SymbolZas);
end;
8: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // l
  TurnLeft;
  SymbolZas.typ := S_NP;
  Pridej_do_zasobniku(SymbolZas);
end;
9: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // r
  TurnRight;
  SymbolZas.typ := S_NP;
  Pridej_do_zasobniku(SymbolZas);
end;
10: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // ]
  Vyjmi_ze_zasobniku(vrchol_zas);              // V
  val := V.atribcislo;
  Vyjmi_ze_zasobniku(vrchol_zas);              // [
```

```
  Vyjmi_ze_zasobniku(vrchol_zas);              // Z
  case vrchol_zas.atribop of
    S_GO: Go(val);
    S_MESSAGE: Message(val);
    S_WAIT: Wait(val);
    else error(....);
  end;
  SymbolZas.typ := S_NP;
  Pridej_do_zasobniku(SymbolZas);
end;
11: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // V
  val := vrchol_zas.atribcislo;
  Vyjmi_ze_zasobniku(vrchol_zas);              // =
  Vyjmi_ze_zasobniku(vrchol_zas);              // i
  Zmen(vrchol_zas.atribstr, val);
  SymbolZas.typ := S_NP;
  Pridej_do_zasobniku(SymbolZas);
end;
12: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // g
  SymbolZas.typ := S_NZ;
  SymbolZas.atribop := S_GO;
  Pridej_do_zasobniku(SymbolZas);
end;
13: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // m
  SymbolZas.typ := S_NZ;
  SymbolZas.atribop := S_MESSAGE;
  Pridej_do_zasobniku(SymbolZas);
end;
14: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // w
  SymbolZas.typ := S_NZ;
  SymbolZas.atribop := S_WAIT;
  Pridej_do_zasobniku(SymbolZas);
end;
15: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // M
  val := vrchol_zas.atribcislo;
  Vyjmi_ze_zasobniku(vrchol_zas);              // A
  SymbolZas.typ := S_NV;
  if vrchol_zas.atribop = S_PLUS then
    SymbolZas.atribcislo :=  vrchol_zas.atribcislo + val
  else if vrchol_zas.atribop = S_MINUS then
    SymbolZas.atribcislo :=  vrchol_zas.atribcislo - val
  else error(...);
  Pridej_do_zasobniku(SymbolZas);
end;
16: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // +
  Vyjmi_ze_zasobniku(vrchol_zas);              // V
```

```
  SymbolZas.typ := S_NA;
  SymbolZas.atribop := S_PLUS;
  SymbolZas.atribcislo := vrchol_zas.atribcislo;
  Pridej_do_zasobniku(SymbolZas);
end;
17: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // -
  Vyjmi_ze_zasobniku(vrchol_zas);              // V
  SymbolZas.typ := S_NA;
  SymbolZas.atribop := S_MINUS;
  SymbolZas.atribcislo := vrchol_zas.atribcislo;
  Pridej_do_zasobniku(SymbolZas);
end;
18: begin
  SymbolZas.typ := S_NA;
  SymbolZas.atribop := S_PLUS;
  SymbolZas.atribcislo := 0;
  Pridej_do_zasobniku(SymbolZas);
end;
19: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // F
  val := vrchol_zas.atribcislo;
  Vyjmi_ze_zasobniku(vrchol_zas);              // B
  SymbolZas.typ := S_NM;
  if vrchol_zas.atribop = S_MUL then
    SymbolZas.atribcislo :=  vrchol_zas.atribcislo * val
  else if vrchol_zas.atribop = S_DIV then begin
    if val=0 then error('Dělení nulou')
      else SymbolZas.atribcislo :=  vrchol_zas.atribcislo / val
  end else error(...);
  Pridej_do_zasobniku(SymbolZas);
end;
20: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // *
  Vyjmi_ze_zasobniku(vrchol_zas);              // M
  SymbolZas.typ := S_NB;
  SymbolZas.atribop := S_MUL;
  SymbolZas.atribcislo := vrchol_zas.atribcislo;
  Pridej_do_zasobniku(SymbolZas);
end;
21: begin
  Vyjmi_ze_zasobniku(vrchol_zas);              // /
  Vyjmi_ze_zasobniku(vrchol_zas);              // M
  SymbolZas.typ := S_NB;
  SymbolZas.atribop := S_DIV;
  SymbolZas.atribcislo := vrchol_zas.atribcislo;
  Pridej_do_zasobniku(SymbolZas);
end;
22: begin
  SymbolZas.typ := S_NB;
  SymbolZas.atribop := S_MUL;
  SymbolZas.atribcislo := 1;
```

```
      Pridej_do_zasobniku(SymbolZas);
    end;
    23: begin
      Vyjmi_ze_zasobniku(vrchol_zas);              // n
      SymbolZas.typ := S_NF;
      SymbolZas.atribcislo := vrchol_zas.atribcislo;
      Pridej_do_zasobniku(SymbolZas);
    end;
    24: begin
      Vyjmi_ze_zasobniku(vrchol_zas);              // i
      SymbolZas.typ := S_NF;
      SymbolZas.atribcislo := ZjistiHodnotu(vrchol_zas.atribstr);
      Pridej_do_zasobniku(SymbolZas);
    end;
    25: begin
      Vyjmi_ze_zasobniku(vrchol_zas);              // )
      Vyjmi_ze_zasobniku(vrchol_zas);              // V
      SymbolZas.typ := S_NF;
      SymbolZas.atribcislo := vrchol_zas.atribcislo;
      Vyjmi_ze_zasobniku(vrchol_zas);              // (
      Pridej_do_zasobniku(SymbolZas);
    end;
end;
```

Zbylé operace v tabulce:

```
procedure error(const hlaska: string);
begin
  Konec := true;
  writeln('Chyba při syntaktické analýze na řádku ',znak.cislo,
    ', sloupci ',znak.pozice,': ',hlaska);
end;

procedure push;
var SymbolZas: TSymbolZas;
begin
  with SymbolZas do begin
    typ := symbol.typ;
    atribcislo := symbol.atribcislo;
    atribstr := symbol.atribstr;
  end;
  Pridej_do_zasobniku(SymbolZas);
  Lex;         // lexikální analyzátor načte další symbol
end;

procedure accept;
begin
  Konec := true;
end;
```

PŘÍLOHA C

Generování kódu assembleru pro výraz

C.1 Popis jazyka

Účelem našeho překladače je přeložit matematický přiřazovací výraz do kódu assembleru. Přeložíme pouze samotný přiřazovací výraz, nebudeme se zabývat adresací, strukturou celého programu ani podrobnostmi práce s proměnnými.

Syntaktická struktura jazyka je následující:

- celý přiřazovací výraz je ve tvaru `proměnná = výraz`,
- proměnné a konstanty jsou celočíselné (nezáporné),
- ve výrazu jsou použity binární operátory $+, -, *, /$ a závorky, operátory jsou s běžnou prioritou.

Sémantika:

- ve výpisu musí být zachovávána priorita operátorů,
- používáme dočasné proměnné, které musí být včas vytvořeny a zařazeny do tabulky symbolů, a také po svém využití zrušeny (odstraněny z tabulky symbolů, týká se dočasných proměnných),
- překladač generuje instrukce ve tvaru podle tabulky C.1,
- operandy instrukcí jsou buď přímé (číslo) nebo nepřímé (název proměnné uzavřený do hranatých závorek) a nebo registr (budeme používat pouze registr `AX`), pro zjednodušení budou všechna čísla i proměnné zabírat paměť o délce 16 bitů (2 Byte).

Instrukce	*Význam*
`MOV dest, source`	přesune obsah druhého parametru do prvního, `dest = source`
`ADD dest, source`	přičte obsah druhého parametru k obsahu prvnímu, `dest = dest + source`
`SUB dest, source`	odečte obsah druhého parametru od prvního, `dest = dest - source`
`MUL source`	vynásobí obsah registru AX parametrem, `AX = AX * source`
`DIV source`	vydělí obsah registru AX parametrem, `AX = AX / source`

Tabulka C.1: Instrukce Assembleru pro aritmetické operace

Je třeba zdůraznit, že při překladu se neprovádí interpretace, pouze vytváříme kód v assembleru, při jehož vyhodnocování je výraz počítán.

C.2 Popis struktury programu – gramatika

Gramatika bude podobná té v příkladu 6.6 na straně 175, ale obohatíme ji o druhou úroveň operátorů a závorky. Sémantika se tím zkomplikuje, ale generovaný kód bude správný.
$S \to i = V$
$V \to AB$
$B \to +AB \mid -AB \mid \varepsilon$
$A \to CD$
$D \to *CD \mid /CD \mid \varepsilon$
$C \to n \mid i \mid (V)$

Kromě běžných matematických operací použijeme tyto sémantické funkce:

- $\mathrm{output}(Op, A_1, A_2)$ vypisující na výstup instrukci assembleru (operátor a dva argumenty),
- $\mathrm{NewTemp}$ je funkce vracející ukazatel na nově vytvořenou dočasnou proměnnou (proměnná je zařazena do tabulky symbolů),
- $\mathrm{DestroyTemp}(Temp)$ odstraní z tabulky symbolů danou proměnnou,
- $\mathrm{VarName}(Var)$ obklopí zadaný název proměnné hranatými závorkami a tak připraví řetězec, který se může stát součástí instrukce na výstup.

Použijeme atributy symbolů $V[dest], A[temp], B[dtemp, dest], C[op, dtemp], D[temp], n[lex], i[lex]$.

Význam atributů je následující:

- $dest$ je dědičný, přenáší se v něm název proměnné, do které má být přiřazen výsledek výpočtu (přiřazení se provede po vyhodnocení celého stromu v pravidle $B \rightarrow \varepsilon$), v uzávorkovaném výrazu je také použit pro odeslání dočasné proměnné, do které je uložen výsledek podvýrazu v závorce,
- $temp$ je syntetizovaný, obsahuje název dočasné proměnné vytvořené v pravidle $D \rightarrow \varepsilon$ a posílá se směrem nahoru po pravidlech se symbolem D, v této dočasné proměnné je uložen výsledek výpočtu podvýrazu s operátory $*$ a $/$,
- $dtemp$ je dědičný, posílá se zleva doprava po pravidlech se symbolem B, obsahuje mezivýsledek výpočtu operátorů $+$ a $-$, u symbolu C se používá v rámci jednoho pravidla,
- lex je syntetizovaný atribut terminálů, obsahuje řetězec s číslem nebo názvem proměnné.

$S \rightarrow i \ \{V.dest = i.name\} \ = V$

$V \rightarrow A \ \{B.dtemp = A.temp, B.dest = V.dest\} \ B$

$B_0 \rightarrow +A \ \{\text{output}('\text{MOV}', '\text{AX}', \text{VarName}(B_0.dtemp))$
$\text{output}('\text{ADD}', '\text{AX}', \text{VarName}(A.temp))$
$\text{output}('\text{MOV}', \text{VarName}(B_0.dtemp), '\text{AX}')$
$\text{DestroyTemp}(A.temp),$
$B_1.dtemp = B_0.dtemp, B_1.dest = B_0.dest\} \ B_1$

$B_0 \rightarrow -A \ \{\text{output}('\text{MOV}', '\text{AX}', \text{VarName}(B_0.dtemp))$
$\text{output}('\text{SUB}', '\text{AX}', \text{VarName}(A.temp))$
$\text{output}('\text{MOV}', \text{VarName}(B_0.dtemp), '\text{AX}')$
$\text{DestroyTemp}(A.temp),$
$B_1.dtemp = B_0.dtemp, B_1.dest = B_0.dest\} \ B_1$

$B \rightarrow \varepsilon \ \{\text{output}('\text{MOV}', '\text{AX}', \text{VarName}(B.dtemp))$
$\text{output}('\text{MOV}', \text{VarName}(B.dest), '\text{AX}')$
$\text{DestroyTemp}(B.dtemp)\}$

$A \rightarrow \{C.op = \text{O\_NEDEF}\} \ CD \ \{A.temp \ D.temp\}$

$D_0 \rightarrow * \ \{C.op = \text{O\_MUL}\} \ CD_1 \ \{D_0.temp = D_1.temp\}$

$D_0 \rightarrow / \ \{C.op = \text{O\_DIV}\} \ CD_1 \ \{D_0.temp = D_1.temp\}$

$D \rightarrow \varepsilon \ \{D.temp = \text{NewTemp}$
$\text{output}('\text{MOV}', \text{VarName}(D.temp), '\text{AX}')\}$

$C \rightarrow n \ \{\text{case} \ C.op \ \text{of}$
$\text{O\_NEDEF} : \text{output}('\text{MOV}', '\text{AX}', n.lex)$
$\text{O\_MUL} : \text{output}('\text{MUL}', n.lex, '')$
$\text{O\_DIV} : \text{output}('\text{DIV}', n.lex, '')\}$

$C \to i$ {case $C.op$ of
O_NEDEF : output('MOV', 'AX', VarName($i.lex$))
O_MUL : output('MUL', VarName($i.lex$), '')
O_DIV : output('DIV', VarName($i.lex$), '')}
$C \to$ ({$C.dtemp$ = NewTemp
output('MOV', VarName($C.dtemp$), 'AX')
$V.dest$ = NewTemp}
V {output('MOV', 'AX', VarName($C.dtemp$))
DestroyTemp($C.dtemp$)
case $C.op$ of
O_NEDEF : output('MOV', 'AX', VarName($V.dest$))
O_MUL : output('MUL', VarName($V.dest$), '')
O_DIV : output('DIV', VarName($V.dest$), '')})

V gramatice jsme splnili požadavek na rušení nepotřebných dočasných proměnných. Každá dočasná proměnná v syntetizovaném atributu $temp$ je zrušena hned, jak je z ní načtena hodnota, dočasné proměnné z atributu $dtemp$ jsou rušeny na konci větve, když je informce v nich uložená využita naposledy.

Gramatika negeneruje zcela optimální kód, lepší by bylo naprogramovat funkci output tak, aby generovala některý intermediální kód (nejlépe 3-adresový) reprezentovaný posloupností binárních záznamů (v souboru nebo v paměti), který by se po optimalizacích přeložil na assembler.

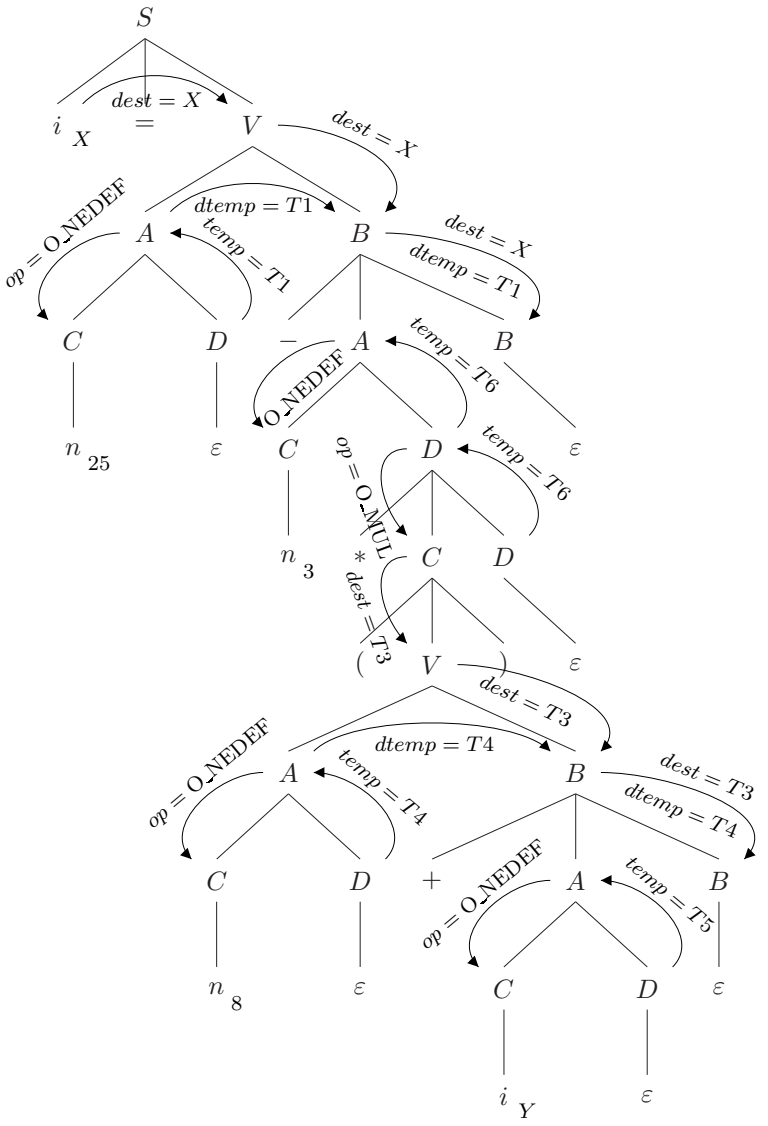
Optimalizaci by si také zasloužil výpočet s operátory + a − v případě, že podstrom symbolu A generuje podvýraz obsahující pouze číslo nebo proměnnou (to by se dalo provést zjištěním hloubky rekurze, viz příklad 5.13 na straně 134).

C.3 Příklad použití

Než začneme programovat, bude lepší se lépe podívat na způsob zpracování vstupu.

Na vstupu je řetězec $X = 25 - 3 * (8 + Y)$. Předpokládejme, že proměnné X a Y jsou zařazeny v tabulce symbolů. Následuje výstup pro daný vstupní řetězec (instrukce jsou očíslovány pro snadnější orientaci), derivační strom se zobrazením toku hodnot atributů pro tento výstup je na obrázku C.1.

```
01   MOV AX, 25          08   MOV [T5], AX       15   MUL [T3]
02   MOV [T1], AX        09   MOV AX, [T4]       16   MOV [T6], AX
03   MOV AX, 3           10   ADD AX, [T5]       17   MOV AX, [T1]
04   MOV [T2], AX        11   MOV [T4], AX       18   SUB AX, [T6]
05   MOV AX, 8           12   MOV AX, [T4]       19   MOV [T1], AX
06   MOV [T4], AX        13   MOV [T3], AX       20   MOV AX, [T1]
07   MOV AX, [Y]         14   MOV AX, [T2]       21   MOV [X], AX
```



Obrázek C.1: Derivační strom se zobrazením toku hodnot atributů pro generování kódu

Při zpracování se provádějí tyto akce (můžeme je sledovat na derivačním stromě s tokem atributů):

	$V.dest = X$	procedura `S`
	$C.op =$ O_NEDEF	procedura `A`
`01`	output('MOV', 'AX', '25')	procedura `C`
	$D.temp =$ NewTemp $= T1$	návrat, procedura `D`
`02`	output('MOV', '[T1]', 'AX')	
	$A.temp = T1$	návrat z `D` do `A`

	$B.dtemp = T1$	návrat z A do V
	$B.dest = X$	
	$C.op =$ O_NEDEF	procedury B, A
03	output('MOV', 'AX', '3')	procedura C
	$C.op =$ O_MUL	návrat do A, dále D
	$C.dtemp =$ NewTemp $= T2$	procedura C, závorka
04	output('MOV', '[T2]', 'AX')	
	$V.dest =$ NewTemp $= T3$	
	$C.op =$ O_NEDEF	procedura A
05	output('MOV', 'AX', '8')	procedura C
	$D.temp =$ NewTemp $= T4$	návrat do A, dále D
06	output('MOV', '[T4]', 'AX')	procedura D
	$A.temp = T4$	návrat do A
	$B.dtemp = T4$	návrat do V
	$B.dest = T3$	
	$C.op =$ O_NEDEF	procedury B, A
07	output('MOV', 'AX', '[Y]')	
	$D.temp =$ NewTemp $= T5$	procedura D
08	output('MOV', '[T5]', 'AX')	
	$A.temp = T5$	návrat do A
09	output('MOV', 'AX', '[T4]')	návrat do B
10	output('ADD', 'AX', '[T5]')	
	DestroyTemp($T5$)	
11	output('MOV', '[T4]', 'AX')	
	$B.dtemp = T4$	rekurzívně procedura B
	$B.dest = T3$	
12	output('MOV', 'AX', '[T4]')	
13	output('MOV', '[T3]', 'AX')	
	DestroyTemp($T4$)	
14	output('MOV', 'AX', '[T2]')	návrat až k C se závorkami
	DestroyTemp($T2$)	
15	output('MUL', '[T3]', '')	
	DestroyTemp($T3$)	
	$D.temp =$ NewTemp $= T6$	zpět do D, pak rekurzívně do D
16	output('MOV', '[T6]', 'AX')	
	$D.temp = T6$	zpět do nadřízené D
	$A.temp = T6$	nahoru do A
17	output('MOV', 'AX', '[T1]')	návrat do B
18	output('SUB', 'AX', '[T6]')	
	DestroyTemp($T6$)	

19 output('MOV', '[T1]', 'AX')
$B.dtemp = T1$ posíláme vnořené proceduře B
$B.dest = X$
20 output('MOV', 'AX', '[T1]')
21 output('MOV', '[X]', 'AX')
DestroyTemp($T1$)

C.4 Implementace

Překlad lze naprogramovat prakticky stejně jako u atributové gramatiky z přílohy A, proto zde pouze naznačíme postup a případné odlišnosti.

Deklarace budou kratší a typů symbolů méně, protože zdrojový jazyk je jednodušší. Do tabulky symbolů budeme ukládat pouze názvy proměnných, nikoliv jejich hodnoty, délku názvu omezíme především z důvodu prostorové složitosti tabulky symbolů.

```
type
  TTypSymbolu = (S_ID, S_NUM, S_PLUS, S_MINUS, S_MUL, S_DIV, S_PRIRAD,
    S_LZAV, S_RZAV, S_ENDOFFILE);
  TTypOperatoru = (O_NEDEF, O_MUL, O_DIV);     // další není třeba předávat
  TRetezecZn = string[25];

  TSymbol = record
    typ:   TTypSymbolu;
    atrib: TRetezecZn;
  end;

  PPolozkaTab = ↑TPolozkaTab;       // pro tabulku symbolů
  TPolozkaTab = record
    nazev:   TRetezecZn;
    dalsi: PPolozkaTab;
  end;

var
  symbol:  TSymbol;
  Tabulka: PPolozkaTab;             // ukazatel na první prvek tabulky
```

Je třeba naprogramovat tyto funkce a procedury:

- `expect(symtyp: TTypSymbolu)`, `S`, `V(dest: TRetezecZn)`,
  `A(var temp: TRetezecZn)`,
  `B(dtemp, dest: TRetezecZn)`,
  `C(op: TTypOperatoru)`,
  `D(var temp: TRetezecZn)`,
- `Lex`,

- `NewTemp() → TRetezecZn, DestroyTemp(t: TRetezecZn)` pro práci s dočasnými proměnnými v tabulce symbolů,
- `output(op, arg1, arg2: TRetezecZn)` pro výstup,
- `Promenna(prom: TRetezecZn) → string` pro úpravu řetězce s názvem proměnné (obklopí hranatými závorkami),
- `error, VypisHodn, VypisTyp` – tytéž jako v předchozích kapitolách.

U procedur pro syntaktickou rekurzi je velmi důležitá posloupnost jednotlivých příkazů včetně volání sémantických funkcí. Ukážeme si kód procedur podle neterminálů.

```
procedure S;
var dest: TRetezecZn;
begin
  if symbol.typ = S_ID then begin
    dest := symbol.atrib;
    expect(S_ID);
    V(dest);
  end else
    error('místo symbolu '+VypisHodn(symbol)+' očekáván '+VypisTyp(S_ID));
end;

procedure V(dest: TRetezecZn);
begin pomtemp: TRetezecZn;
begin
  if symbol.typ in [S_NUM, S_ID, S_LZAV] then begin
    A(pomtemp);
    B(pomtemp, dest);
  end else error(...);       // ošetření chyby podobně jako předchozí,
end;                         // totéž platí i pro následující

procedure B(dtemp, dest: TRetezecZn);
var pomtemp: TRetezecZn;
begin
  case symbol.typ of
    S_PLUS: begin
      expect(S_PLUS);
      A(pomtemp);   // v pomtemp je název dočasné proměnné z podstromu A
      output('MOV', 'AX', Promenna(dtemp));
      output('ADD', 'AX', Promenna(pomtemp));
      output('MOV', Promenna(dtemp), 'AX');
      DestroyTemp(pomtemp);
      B(dtemp, dest);
    end;
    S_MINUS: begin
      expect(S_MINUS);
      A(pomtemp);
      output('MOV', 'AX', Promenna(dtemp));
```

```
      output('SUB', 'AX', Promenna(pomtemp));
      output('MOV', Promenna(dtemp), 'AX');
      DestroyTemp(pomtemp);
      B(dtemp, dest);
    end;
    S_RZAV, S_ENDOFFILE: begin
      output('MOV', 'AX', Promenna(dtemp));
      output('MOV', Promenna(dest), 'AX');
      DestroyTemp(dtemp);
    end;
    else error(...);
  end;
end;

procedure A(var temp: TRetezecZn);
begin
  if symbol.typ in [S_NUM, S_ID, S_LZAV] then begin
    C(0_NEDEF);
    D(temp);
  end else error(...);
end;

procedure D(var temp: TRetezecZn);
begin
  case symbol.typ of
    S_MUL: begin
      C(O_MUL);
      D(temp);
    end;
    S_DIV: begin
      C(O_DIV);
      D(temp);
    end;
    S_PLUS,S_MINUS,S_RZAV,S_ENDOFFILE: begin
      temp := NewTemp;
      output('MOV', Promenna(temp), 'AX');
    end;
    else error(...);
  end;
end;

procedure C(op: TTypOperatoru);
var dtemp, dest: TRetezecZn;
begin
  case symbol.typ of
    S_NUM,S_ID: begin
      case op of
        O_NEDEF: output('MOV', 'AX', Promenna(symbol.atrib));
        O_MUL:   output('MUL', Promenna(symbol.atrib), '');
```

```
        O_DIV:   output('DIV', Promenna(symbol.atrib), '');
      end;
      expect(symbol.typ);
    end;
    S_LZAV: begin
      expect(S_LZAV);
      dtemp := NewTemp;
      output('MOV', Promenna(dtemp), 'AX');
      dest := NewTemp;
      V(dest);
      output('MOV', 'AX', dtemp);
      DestroyTemp(dtemp);
      case op of
        O_NEDEF: output('MOV', 'AX', Promenna(dest));
        O_MUL:   output('MUL', Promenna(dest), '');
        O_DIV:   output('DIV', Promenna(dest), '');
      end;
      DestroyTemp(dest);
      expect(S_RZAV);
    end;
    else error(...);
  end;
end;
```

Seznam doporučené literatury

[1] AABY, A. A.: *Compiler Construction using Flex and Bison* [online]. 1996.
URL: http://foja.dcs.fmph.uniba.sk/kompilatory/docs/compiler.pdf [cit. 2008-10-2]

[2] ABELSON, H. – SUSSMAN, G. J. – SUSSMAN, J.: *Structure and Interpretation of Computer Programs* [online]. 2nd edition. Cambridge (Massachusetts), London (England): MIT Press, 1993.
URL: http://mitpress.mit.edu/sicp/full-text/book/book.html [cit. 2008-7-1]

[3] AHO, A. V. – SETHI, R. – ULLMAN, J. D.: *Compilers: Principles, Techniques and Tools*. Boston, MA (USA): Addison-Wesley, 1986 (tisk 2006).
ISBN: 978-0321486813.

[4] ČEŠKA, M. – BENEŠ, M. – HRUŠKA, T.: *Překladače* [online]. Skriptum VUT. Brno: VUT, Fakulta elektrotechnická, 1993.
ISBN: 80-214-0491-4.
URL: http://www.fit.vutbr.cz/~meduna/fjp/skripta.pdf [cit. 2008-7-1]

[5] ČEŠKA, M. – MELICHAR, B. – RICHTA, K.: *Konstrukce překladačů*. Skriptum ČVUT. Praha: Vydavatelství ČVUT, 2000.
ISBN: 80-01-02028-2.

[6] German National Research Center for Information Technology: *The Catalog of Compiler Construction Tools* [online]. 2006. Copyright © 1996–2006.
URL: http://catalog.compilertools.net/ [cit. 2008-7-1]

[7] Google Directory: *Compilers* [online].
URL: http://www.google.com/Top/Computers/Programming/Compilers/ [cit. 2008-7-1]

[8] CHYTIL, M.: *Automaty a gramatiky*. Praha: SNTL, 1984.

[9] MEDUNA, A.: *Automata and Languages: Theory and Application*. Springer Verlag, 2000. ISBN: 978-1-85233-074-3.
Dostupné na Google Books: http://books.google.cz, jako klíčová slova zadejte celý název knihy [cit. 2008-7-1]

[10] MELICHAR, B.: *Jazyky a překlady*. Praha: Vydavatelství ČVUT, 1999. ISBN: 80-01-01511-4.

[11] MELICHAR, B.: *Základy překladačů*. Praha: Vydavatelství ČVUT, 1989.

[12] PALETA, P.: *Co programátory ve škole neučí*. Brno: Computer Press, 2004. ISBN: 80-251-0073-1.

[13] RAYMOND, E. S.: *Umění programování v Unixu*. Brno: Computer Press, 2004. ISBN: 80-251-0225-4.

[14] TERRY, P. D.: *Compilers and Compiler Generators: an introduction with C++* [online]. Rhodes University, Grahamstown (South Africa), 1996 (poslední revize 2005). URL: http://scifac.ru.ac.za/compilers/ [cit. 2008-7-4]

[15] TERRY, P. D.: *Compiling with C# and Java*. Addison Wesley, 2004. ISBN: 978-0321263605.

[16] VIRIUS, M.: *Jazyky C a C++: Kompletní kapesní průvodce*. Praha: Grada, 2005. ISBN: 80-247-1494-9.

[17] WIRTH, N.: *Algoritmy a štruktůry údajov*. Bratislava (Slovensko): Alfa, 1989. ISBN: 80-05-00153-3.

Rejstřík