Šárka Vavrečková

Programování překladačů

druhá upravená verze

Opava
Poslední aktualizace: 9. prosince 2023

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

Programování překladačů
druhá upravená verze

RNDr. Šárka Vavrečková, Ph.D.

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

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

Sázeno v systému LaTeX

Předmluva

Co najdeme v těchto skriptech

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

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

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

Značení

Ve skriptech se používají následující barevné ikony:

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

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

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

Definice

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

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

Příklad

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

Úkol

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

Obsah

Předmluva **iii**

Úvod **1**

1 Překladač a jeho struktura **3**
1.1 Základní pojmy 3
1.2 Hlavní části překladače 5
1.3 Průchody překladače 6
1.4 Konverzační překladače 8
1.5 Zpracování chyb 8
1.6 Překladače ve vztahu k jiným programům 9
1.6.1 Generátory překladačů 9
1.6.2 Aplikace pro jinou platformu 10
1.6.3 Portování 10
1.7 Editory pro překladače 11

2 Lexikální analýza **15**
2.1 Popis lexikální struktury jazyka 15
2.2 Rozpoznávání symbolů 18
2.3 Implementace 19
2.3.1 Vstup a výstup lexikálního analyzátoru 20
2.3.2 Metody pro konečné a nekonečné jazyky 22
2.3.3 Uplatnění metod na zvolený jazyk 27

Přílohy **32**

Úvod

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

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

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

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

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

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

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

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

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

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

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

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

Překladač a jeho struktura

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

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

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

1.1 Základní pojmy

Definice (Překladač)

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

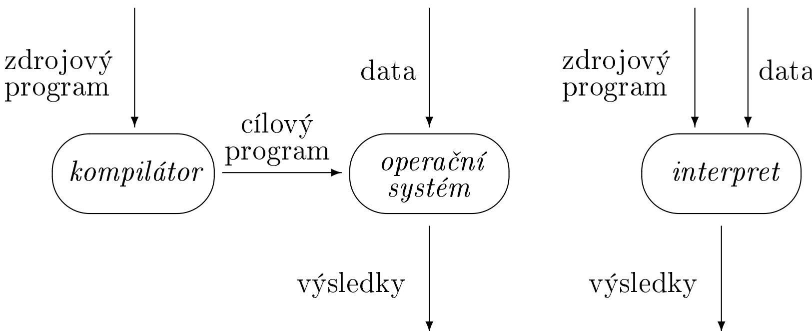
Podle typu cílového programu rozlišujeme tyto druhy překladačů:

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

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

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

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



Obrázek 1.1: Schéma kompilačního a interpretačního překladače

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

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

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

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

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

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

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

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

Tabulka 1.1: Srovnání vlastností kompilačního a interpretačního překladače

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

1.2 Hlavní části překladače

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

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

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

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

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

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

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

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

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

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

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

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

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

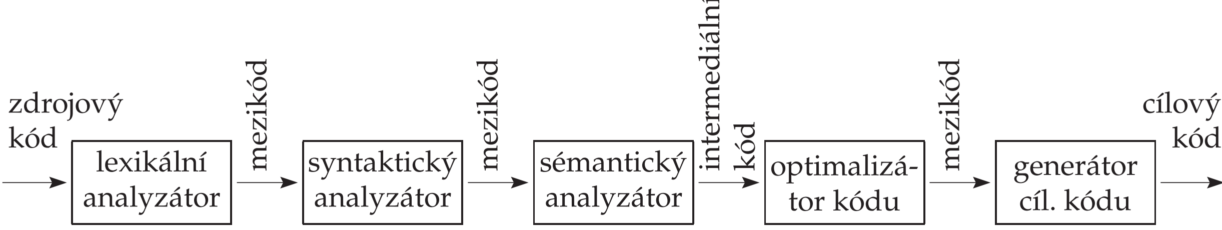
Přední část provádí analýzu („rozpitvává vstup“), zadní provádí syntézu (dává dohromady výstup).

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

1.3 Průchody překladače

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

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

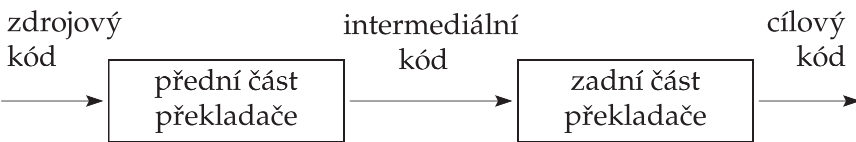


Obrázek 1.2: Víceprůchodový překladač, každá fáze v samostatném průchodu

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

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

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



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

Víceprůchodový překladač má tyto výhody:

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

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

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

1.4 Konverzační překladače

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

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

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

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

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

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

1.5 Zpracování chyb

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

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

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

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

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

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

✂ Překladač může na chybu reagovat dvěma způsoby:

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

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

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

✎ Chyby na straně uživatele překladače dělíme do tří kategorií:

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

1.6 Překladače ve vztahu k jiným programům

1.6.1 Generátory překladačů

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

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

Každý překladač je charakterizován třemi jazyky:

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

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

Další informace

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

1.6.2 Aplikace pro jinou platformu

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

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

1.6.3 Portování

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

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

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

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

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

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

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

1.7 Editory pro překladače

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

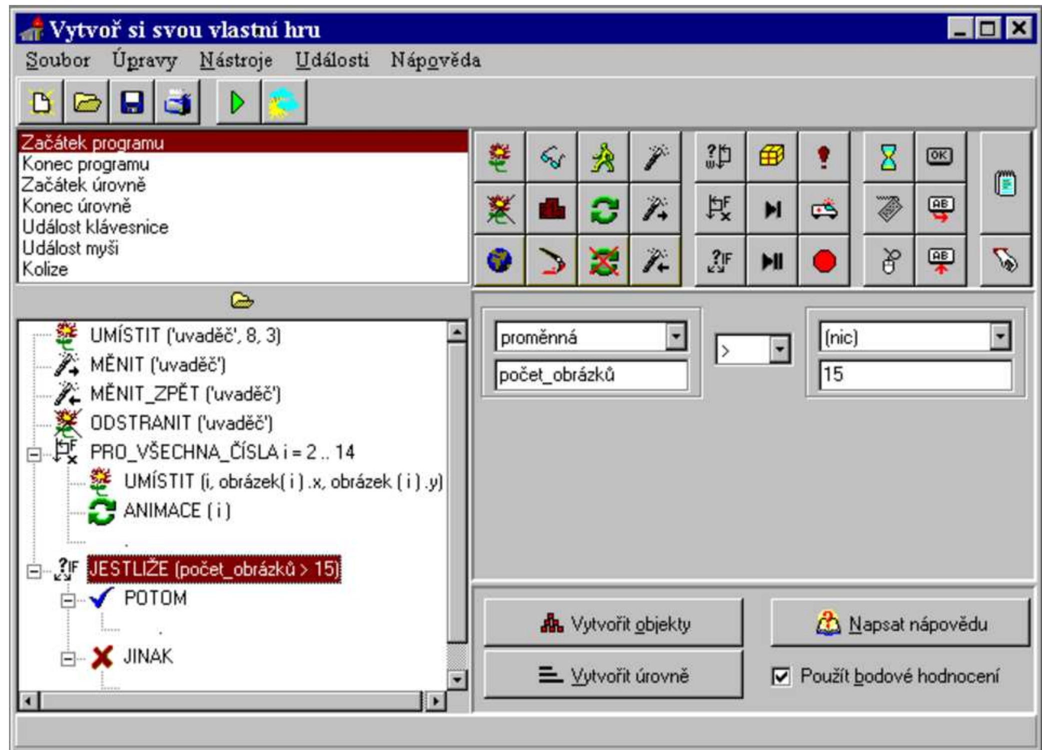
Editor dodávaný s překladačem bývá zpravidla napsán v zdrojovém jazyce, který přijímá jeho překladač (a také bývá tímto překladačem přeložen), aby autor demonstroval použitelnost jazyka a překladače.

Tyto editory mohou být realizovány několika způsoby:

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

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

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



Obrázek 1.4: Ukázka prostředí grafického editoru určeného pro děti

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

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

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

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

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

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

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

Další informace

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

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

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

Úkoly

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

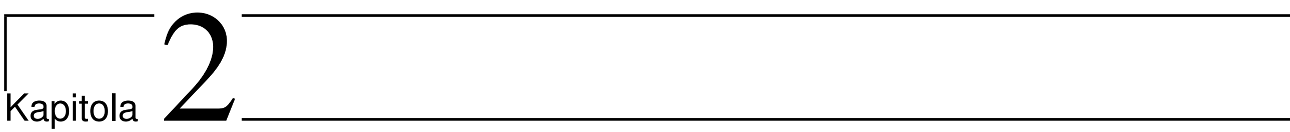
[4] RAD – Rapid Application Development, rychlý vývoj aplikací, je trend pro vývojová prostředí, kdy alespoň část GUI vyvíjené aplikace programátor určuje rychle „pomocí myši".

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

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

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

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



Lexikální analýza

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

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

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

2.1 Popis lexikální struktury jazyka

V následující tabulce jsou shrnuty nejdůležitější vlastnosti lexikální analýzy.

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

Tabulka 2.1: Vlastnosti lexikální analýzy

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

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

Obvykle se používají tyto vstupní formáty:

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

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

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

Příklad

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

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

Výstup lexikálního analyzátoru je tento soubor:

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

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

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

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

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

Příklad

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

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

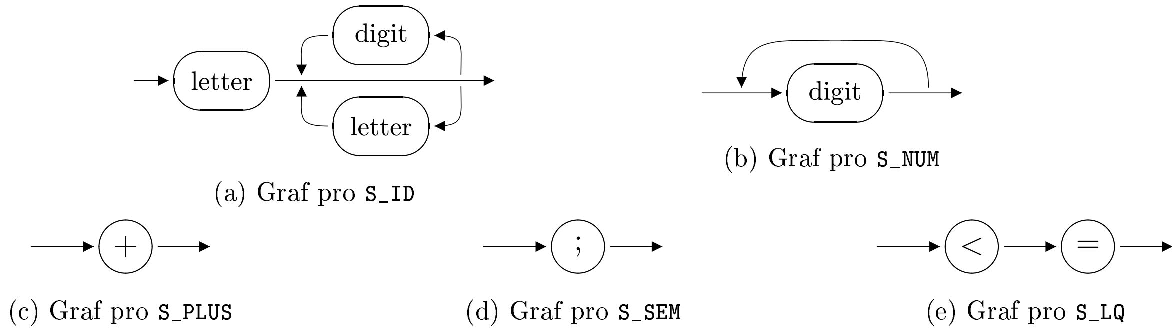
Z abstraktní báze se posunujeme ke konkrétní reprezentaci struktury symbolů. Můžeme použít *syntaktické grafy* nebo přímo pravidla regulární gramatiky.

Příklad

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

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

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



(a) Graf pro `S_ID` (b) Graf pro `S_NUM`

(c) Graf pro `S_PLUS` (d) Graf pro `S_SEM` (e) Graf pro `S_LQ`

Obrázek 2.1: Syntaktické grafy některých symbolů

$$
\begin{array}{lcll}
S & \rightarrow & l \mid lA & \text{identifikátory} \\
A & \rightarrow & l \mid d \mid lA \mid dA & \\
S & \rightarrow & d \mid dB & \text{čísla} \\
B & \rightarrow & d \mid dB & \\
S & \rightarrow & + \mid - \mid * \mid / & \text{aritmetické operátory} \\
S & \rightarrow & > \mid < \mid = \mid <C \mid >D & \text{relační operátory} \\
C & \rightarrow & = \mid > & \\
D & \rightarrow & = & \\
S & \rightarrow & :E & \text{operátor přiřazení} \\
E & \rightarrow & = & \\
S & \rightarrow & (\mid) \mid ; & \text{pomocné symboly} \\
S & \rightarrow & mezera \mid konec_řádku &
\end{array}
$$

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

2.2 Rozpoznávání symbolů

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

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

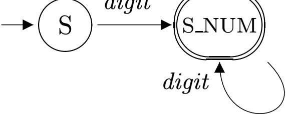
Kdybychom pro rozpoznávání symbolů na vstupu použili „klasické“ funkce typu `strcmp`, dostaneme se především u rozsáhlejšího překladače do velkých problémů: při každém porovnávání bychom se totiž opakovaně vraceli na začátek symbolu, tentýž znak bychom museli opakovaně zpracovávat hlavně při

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

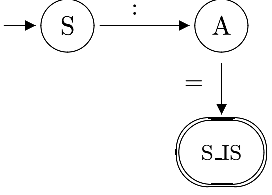
Příklad

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

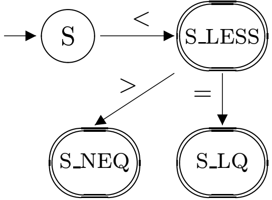
(a) Konečný automat pro `S_ID`



(b) Konečný automat pro `S_NUM`



(c) Konečný automat pro symbol „: =“



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

Obrázek 2.2: Konečný automat pro některé symboly

Ostatní diagramy jsou podobné, jejich vytvoření necháváme na čtenáři.

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

2.3 Implementace

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

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

V předchozí sekci jsme vytvořili konečný automat rozpoznávající zdrojový jazyk překladače. Nyní

sestavíme program, který realizuje výpočet tohoto automatu.

Budeme postupovat takto:

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

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

2.3.1 Vstup a výstup lexikálního analyzátoru

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

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

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

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

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

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

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

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

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

Pro uschování načtené části vstupu zvolíme tuto strukturu:

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

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

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

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

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

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

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

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

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

A) Přímé stavové programování

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

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

a_0 A a_1 a_2 a_n

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

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

S < S_LESS > = S_NEQ S_LQ

```
      default: symbol.typ = S_LESS;
    }
    ...
  default: vypis_chybu(...);  // pravděpodobně takový znak, který zde nemá co dělat
}
```

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

B) Tabulka přechodů jako celočíselná matice

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

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

Příklad

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

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

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

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

Vytvoříme si konstanty pro chybový a koncové stavy, aby byl následující kód přehlednější. Tabulku budeme v paměti reprezentovat formou 2D pole, prvky budou typu `int`. Protože indexy pole mají být v jazyce C celočíselné, můžeme buď použít ASCII hodnoty (ale to by znamenalo hodně „širokou“ tabulku), nebo si naprogramujeme konverzní funkci, která jednotlivým znakům přiřadí číslo, v našem případě z intervalu 0–7. Obě možnosti jsou použitelné, záleží na jazyce. Případně můžeme použít omezený interval z ASCII, podle znaků povolených v názvech proměnných a jiných identifikátorů.

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

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

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

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

  tab[0][0]=1;       tab[0][2]=2;       tab[0][4]=5;
  tab[1][1]=k_if;
```

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

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

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

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

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

C) Stav reprezentován proměnnou

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

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

Příklad

Automat z předchozího příkladu přepíšeme takto:

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

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

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

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

2.3.3 Uplatnění metod na zvolený jazyk

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

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

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

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

void lex() {        // načte jeden symbol do globální proměnné symbol
// funkce DejZnak() byla už volána, máme přednačtený znak v proměnné vstup
  symbol.atrib = "";
  while (vstup.znak == ' ' || vstup.znak == '\t')
    DejZnak();                                   // přeskočíme mezery a tabulátory

if (vstup.znak >= 'A' && vstup.znak <= 'Z') {   // začíná písmenem?
    do {
      symbol.atrib += vstup.znak;
      DejZnak();
    } while (vstup.znak >= 'A' && vstup.znak <= 'Z');  // příp. přidejte podtržítko
    symbol.typ = S_ID;
    lex_klic_slova(symbol.atrib);
  }

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

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

        '=':
          DejZnak();
          symbol.typ = S_LQ;           // <=
          break;
      default: symbol.typ = S_LESS;    // <
    };
```

```
    ...                                        // podobně všechny ostatní symboly
    default: ...                              // ošetření chyby
  }
}
```

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

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

Příklad

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

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

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

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

```
const int
  k_chyba = 23,    k_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.

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

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

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

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

Tabulka 2.3: Tabulka přechodů pro klíčová slova zvoleného jazyka

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

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

void lex_klic_slova(string s) {
int
  stav   = 0,          // aktuální stav automatu
  pozice = 1,          // pozice v testovaném řetězci s
```

```
  delka  = length(s), // délka řetězce s
  znak;               // číslo znaku podle seznamu znaků klíčových slov

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

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

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

Úkoly

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

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

[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.

Automat má rozpoznávat jednotlivá slova jazyka, bude mít pro každé slovo jiný koncový stav. Gramatiku vytvořte tak, aby bylo možné konstruovat automat přímo jako deterministický, bez nutnosti další transformace.

5. Naprogramujte konečný automat z úkolu 4 některou z metod z této kapitoly nebo jejich kombinací (metody jsou popsány v podkapitole 2.3.2 od strany 22, možnost kombinace metod v podkapitole 2.3.3 od strany 27).
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 27).
9. Upravte kód metody přímého stavového programování použitý na načítání čísel (celý kód začíná na straně 27) tak, aby lexikální analyzátor převáděl načtené číslo z řetězcové na číselnou reprezentaci, a to bez použití funkcí poskytovaných programovacím jazykem, ve kterém pracujete. Pro celé číslo bude v symbolu uložena jak jeho číselná hodnota, tak i řetězcové vyjádření. Symbol ukládejte do proměnné následujícího datového typu:

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

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

Přílohy