EVROPSKÁ UNIE

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

OP Vzdělávání pro konkurenceschopnost

UNIVERSITAS OSTRAVIENSIS

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

PŘEKLADAČE

URČENO PRO VZDĚLÁVÁNÍ V AKREDITOVANÝCH STUDIJNÍCH PROGRAMECH

HASHIM HABIBALLA

ČÍSLO OPERAČNÍHO PROGRAMU: CZ.1.07
NÁZEV OPERAČNÍHO PROGRAMU:
VZDĚLÁVÁNÍ PRO KONKURENCESCHOPNOST
OPATŘENÍ: 7.2
ČÍSLO OBLASTI PODPORY: 7.2.2

INOVACE VÝUKY INFORMATICKÝCH PŘEDMĚTŮ VE STUDIJNÍCH PROGRAMECH OSTRAVSKÉ UNIVERZITY

REGISTRAČNÍ ČÍSLO PROJEKTU: CZ.1.07/2.2.00/28.0245

OSTRAVA 2013

Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky

Recenzent: Doc. RNDr. PaedDr. Eva Volná, PhD.

Název:	Překladače
Autor:	Doc. RNDr. PaedDr. Hashim Habiballa, PhD., Ph.D.
Vydání:	druhé (inovované), 2014
Počet stran:	99

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídá autor.

© Hashim Habiballa
© Ostravská univerzita v Ostravě

OBSAH

1 POJEM PŘEKLADAČE 6

1.1 HISTORIE PŘEKLADAČŮ 8

1.2 PŘEKLADAČ A JEHO ZÁKLADNÍ TYPY 10

2 STRUKTURA PŘEKLADAČE 13

2.1 ANALYTICKÁ ČÁST PŘEKLADAČE 14

2.2 SYNTETICKÁ ČÁST PŘEKLADAČE 15

2.3 TABULKA SYMBOLŮ 17

2.4 INTERMEDIÁRNÍ KÓD 18

3 LEXIKÁLNÍ ANALÝZA 21

3.1 LEXIKÁLNÍ STRUKTURA JAZYKA 21

3.2 ROZPOZNÁVÁNÍ SYMBOLŮ 22

4 SYNTAKTICKÁ ANALÝZA 25

4.1 ZÁKLADNÍ POJMY 26

4.2 SYNTAKTICKÁ ANALÝZA SHORA DOLŮ 27

4.3 SYNTAKTICKÁ ANALÝZA ZDOLA NAHORU 28

4.4 BACKUSOVA-NAUROVA FORMA 30

4.5 METODA REKURZIVNÍHO SESTUPU 31

4.6 PŘÍMÁ IMPLEMENTACE ANALYTICKÉ ČÁSTI PŘEKLADAČE 35

5 AUTOMATIZOVANÉ NÁSTROJE 41

6 FLEX 44

6.1 INSTALACE 45

6.2 VOLITELNÉ PARAMETRY 45

6.3 FORMÁT VSTUPNÍHO SOUBORU 46

6.4 DEFINIČNÍ ČÁST 49

6.5 ČÁST PRAVIDEL 50

6.6 POROVNÁVÁNÍ VSTUPNÍHO TEXTU 52

6.7 AKCE 53

6.8 PRÁCE SE STAVY 56

7 BISON 59

7.1 INSTALACE 60

7.2 VOLITELNÉ PARAMETRY 60

7.3 FORMÁT VSTUPNÍHO SOUBORU 61

7.4 SYMBOLY ... 62
7.5 SÉMANTIKA JAZYKA ... 62
7.6 DEFINIČNÍ ČÁST ... 63
7.7 ČÁST PRAVIDEL ... 65
7.8 AKCE ... 66
7.9 POZICE SYMBOLŮ ... 68
7.10 PRACOVNÍ ALGORITMUS ... 71
8 PŘÍPADOVÁ STUDIE – REPREZENTACE A PŘEKLAD LOGICKÝCH VÝRAZŮ ... 75
8.1 KONSTRUKCE STROMU ... 78
8.2 OPTIMALIZACE VÝRAZU ... 82
9 LITERATURA ... 98

1 Pojem překladače

Cíl:

Po prostudování této kapitoly pochopíte:

- **co je překladač,**
- **kde se používá v reálných aplikacích,**
- **jaké základní typy používáme,**
- **historický vývoj překladačů.**

Klíčová slova této kapitoly:

Teorie formálních jazyků, překladač, formální překlad.

Průvodce studiem

Tento text volně navazuje na studijní oporu Gramatiky a jazyky. Věnuje se především aplikaci poznatků teorie formálních jazyků a automatů při tvorbě překladačů. Překladače tvoří součást každodenní práce informatika – vývojáře aplikací. Kdykoliv vyvstane potřeba navrhnout a implementovat informační systém, musíme vytvořený zdrojový kód jistého programovacího jazyka aplikace nějakým způsobem převést na požadovaný cílový kód – například spustitelný strojový kód procesoru. Text je aplikovaně orientován, i když obsahuje rovněž jisté množství teoretických konstrukcí, zaměřuje se především na postupy tvorby jednotlivých částí překladače, konkrétní metody implementace a nezbytné pojmy teorie překladačů.

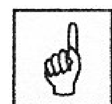
Pojem překladače

Při studiu opory Gramatiky a jazyky jste se seznámili se základy teorie formálních jazyků a poznali jste formální nástroje pro zápis struktury jazyků, jejich vlastnosti, třídy různě složitých jazyků. Dále jste poznali nástroje pro rozpoznávání, zda slovo (jistý kód) vyhovuje specifikaci jazyka – automat. Při studiu základů teorie formálních jazyků a zejména dvou nejjednodušších tříd jazyků – regulárních a bezkontextových – jste se setkali s duálním konceptem gramatiky a automatu k danému jazyku. Gramatika umožňuje generovat daný jazyk (tedy jednotlivé prvky jazyka) a automat naopak rozpoznávat, zda testovaný prvek patří do jazyka. Z tohoto teoretického pohledu může někdy zůstat v pozadí fakt, že tento duální koncept znáte přímo ze své praxe informatiků. Pravděpodobně nejbližším vám bude příklad z oblasti programování a tedy programovacích jazyků. Pokud se učíte používat daný programovací jazyk, učíte se zejména správně zapisovat programy dle jeho specifikace (odhlédneme-li nyní od toho, že chcete, aby program dělal to, co požadujete – to je vyšší stupeň). Učíte se tedy správně zapisovat **syntaxi** daného programovacího jazyka (tedy jeho strukturu z hlediska jazykových vyjadřovacích prostředků). Například u jazyka Pascal víte, že musí obsahovat nejprve deklarace typů, proměnných, dále deklarace funkcí a procedur a nakonec samotnou výkonnou část s popisem algoritmu –programu. Nebo na mnohem nižší úrovni víte, že aritmetický výraz vložený do přiřazovacího příkazu se může skládat s podvýrazů vzájemně spojených operátory sčítaní, odčítání apod. Přičemž nejjednodušším operandem může být kupříkladu celé číslo a důležité je, že tyto výrazy se mohou do sebe vzájemně vnořovat, čímž můžete vytvářet potenciálně nekonečně složité vnořené výrazy.

Syntaxe

Gramatika

Překladač



Toto je vlastně malá část oné syntaxe jazyka, kterou musíte zvládnout. Když uděláme analogii s vašimi teoretickými poznatky z předchozího studia, učíte se vlastně **gramatiku** daného jazyka. Co však s oním duálním konceptem automatu? I on je vaší práci zcela přirozeně přítomen. Po správném napsaní programu samozřejmě vaše práce nekončí. Musíte si zdrojový kód programu pomocí zvoleného **překladače** (kompilátoru) přeložit do formy spustitelného nebo jiného cílového kódu. Součástí každého překladače musí být (mimo jiných mnoha dalších kroků) kontrola, zda je váš program správně zapsán

podle specifikace jazyka. Tuto kontrolu musí provést jistá část překladače – algoritmu, která z teoretického hlediska funguje jako **automat**. Dá vám odpověď, zda je váš program správně napsán – tedy zda zdrojový kód patří do jazyka Pascal.

To je však pouze jeden z nutných činností překladače. Dalším nezbytným krokem jsou kontroly sémantiky kódu – tedy zda kód neobsahuje nějakou logickou chybu, například přiřazování nesprávného typu hodnoty do proměnné (výraz typu string do proměnné typu integer). Dále musí překladač vygenerovat nějakou vnitřní reprezentaci zdrojového kódu a tu pak optimalizovat (například odstranit přiřazování do proměnných, které se již nikde v programu nepoužijí). Nakonec musí být tato vnitřní reprezentace přeložena do požadovaného cílového kódu.

1.1 Historie překladačů

FORTRAN

V angličtině se pro pojem překládání v oblasti programovacích jazyků používá termín „compilation", což v češtině znamená spíše skládání, nebo shromaždování. Pojem „kompilátor" zavedl na začátku padesátých let Grace Murray Hopper. Překlad se totiž prováděl jako skládání sekvencí podprogramu ve strojovém kódu, které byly uloženy v knihovně. Jedním z prvních skutečných překladačů byl dobře známý FORTRAN. Jazyky tehdejší doby byly limitovány hlavně výpočetní kapacitou soudobých počítačů. Optimalizace a efektivita algoritmu zde byla hlavní prioritou. Počítače této doby by se daly přirovnat k dnešním jednodušším programovatelným kalkulátorům. Ovšem rozdíl velikostí je nesrovnatelný.

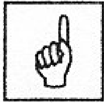
ALGOL 60

Zlom v podstatě překladu nastal v šedesátých letech s příchodem jazyka Algol 60. Ten zavedl tzv. problémově orientovaný přístup tj. důležitým faktorem

bylo to, co je potřeba vyřešit, narozdíl od přístupu stávajících překladačů, které spíše umožňovaly jen přehlednější zápis strojového kódu. Problémově orientovaný přístup je hlavním rysem dnešních jazyku vyšší úrovně. Už jazyky Pascal, Modula a Ada byly vytvořeny nezávisle na konkrétní architektuře. Oddělení od skutečné architektury počítače je dnes ještě umocněno vytvořením virtuálních počítačů, na kterých už jsou provozovány konečné aplikace. Takovýto přístup je charakteristický pro platformy Java nebo .NET.

Dá se říci, že pracujeme-li s počítačem, překladač na nás číhá téměř na každém kroku. Ani si neuvědomujeme, kolik překladů se v počítači děje jen při psaní obyčejného dokumentu. Program rozpoznává jednotlivá slova, snaží se je dle nějakých pravidel uspořádat a také kontroluje správnost jejich zadání. V případe potřeby je dokonce dle pravidel gramatiky jazyka rozděluje nebo naopak spojuje do celku. Když už dokument vytvoříme a chceme jej vytisknout, opět nastává překlad. Dokument je převeden do jazyka tiskárny. U profesionálních tiskáren je většinou implementován jazyk Postscript, kdy do tiskárny je poslán program, který je v ní přeložen a na základě něj je vytvořen finální obraz a spuštěn tisk. V poslední době je ale pro formátování tisku využíváno výkonu počítače a tak do tiskárny je posílán již hotový obraz. Další velmi významnou roli hrají překladače v prostředí webových stránek, kdy je přenášena pouze jednoduchá textová reprezentace s definicemi říkajícími co, kde a jakým způsobem zobrazit a vytvoření podstatně složitější grafické reprezentace je provedeno až při zpracování textu v prohlížeči.

1.2 Překladač a jeho základní typy



Překladač

Vezměme zdrojový (vstupní) jazyk J_Z a cílový (výstupní) jazyk J_C. Překladač analyzuje vstupní program P_Z v jazyce J_Z a vytvoří k němu výstupní program P_C v jazyce J_C.

V závislosti na typu výstupního programu P_C rozdělujeme překladače do dvou kategorií:

Kompilátor

Interpret

- **kompilační překladač** je překladač, který vytvoří ze zdrojového programu ve vyšším programovacím jazyce výstupní program ve strojovém kódu počítače,
- **interpretační překladač** naproti tomu žádný výstupní program nevytváří, ale přímo interpretuje příkazy zdrojového jazyka a provádí příslušné akce.

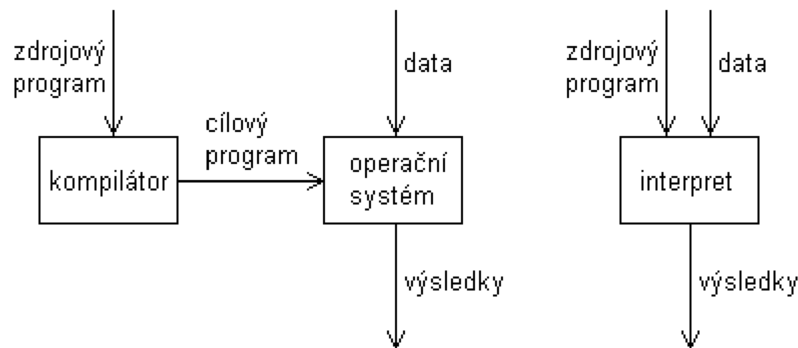


Schéma kompilačního a interpretačního překladače

Nabízí se otázka, proč si nevystačíme pouze s jedním z uvedených přístupů k překladu. Praxe ukázala, že v různých situacích jsou kladeny na překladač různé nároky. Pokud spouštíme program na lokálním počítači, očekáváme od něj mimo jiné také rychlost. A zde se objevuje první rozdíl mezi kompilací a interpretací. U kompilace se provádí analýza a překlad zdrojového programu pouze jednou a nezávisle na samotném spouštění programu, díky čemuž lze provádět např. i časově náročnější kontroly. Uživatel dostává výsledný program ve strojovém kódu, jehož provádění je znatelně rychlejší než

interpretace. Interpret analyzuje příkaz zdrojového souboru pokaždé, když na něj narazí a tedy i výsledná rychlost interpretovaného programu je nižší. Navíc je v případě interpretačního překladače nutné mít při běhu překládaného programu v paměti také samotný překladač, což zvyšuje paměťové nároky spouštěného programu. Těžko bychom však používali interpretační překladače, pokud by neměly oproti kompilátorům také nějaké výhody.

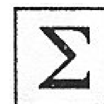
Jednou z velkých výhod interpretačních překladačů je přenositelnost mezi platformami, neboť negenerují strojový kód, který je závislý na konkrétní platformě. Lze mít nainstalován interpretační překladač téhož zdrojového jazyka na různých platformách a všechny překladače přijmou tentýž zdrojový program. Další výhodou interpretů, kterou oceňují především samotní programátoři, je snadnější odhalování chyb a nevyžadování znalosti assembleru (strojového jazyka). Následující tabulka přehledně shrnuje vlastnosti obou typů překladačů. Symbolem * je vždy označen překladač, který daný požadavek zvládá lépe.

Požadavek	Kompilátor	Interpret
Rychlost cílového programu	*	
Přenositelnost zdrojového kódu		*
Paměťové nároky programu	*	
Detekce chyb		*

Srovnání vlastností kompilačního a interpretačního překladače

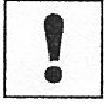
V praxi se však často oba typy překladačů vzájemně překrývají a vznikají tak překladače, které s úspěchem využívají výhod obou typů.

Nejdůležitější probrané pojmy:



překladač
zdrojový a cílový program
interpretační překladač

kompilační překladač



Úkoly k textu:

Pokuste se vyjmenovat pět příkladů překladačů, které používáte a mezi tím, alespoň dva, které nepřekládají programovací jazyky.
Vzpomeňte si alespoň na jeden příklad kompilačního překladače a na jeden příklad interpretačního překladače.

2 Struktura překladače

Cíl:

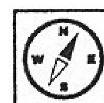
Po prostudování této kapitoly pochopíte:

- z čeho se překladač skládá,
- jak postupně probíhá překlad,
- jakou složitost mají analytické části překladače,
- jakou roli hraje tzv. tabulka symbolů.

Průvodce studiem

Překlad zdrojového programu probíhá v několika fázích. Celý postup lze rozdělit na část analytickou a část syntetickou.

Samotný překladač se skládá z několika komponent, které jsou využívány pro jednotlivé činnosti:



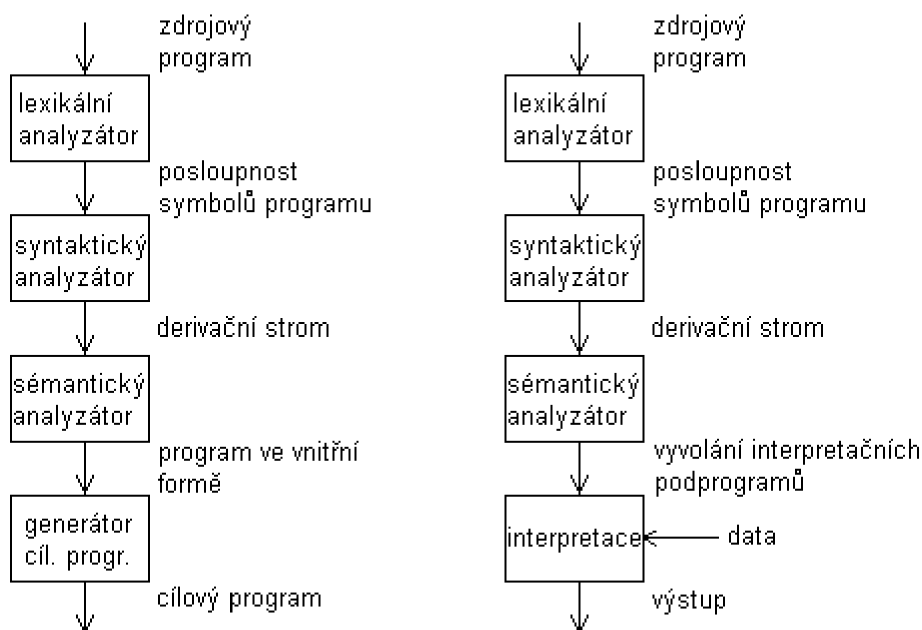
lexikální analyzátor

syntaktický analyzátor

sémantický analyzátor

generování cílového programu / interpretace

Struktura překladače



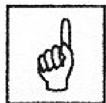
Struktura kompilátoru a interpretu

Podle závislosti na typu cílového kódu lze překladač rozdělit na dvě části:

přední část zahrnuje lexikální, syntaktickou a sémantickou analýzu. Je nezávislá na cílovém kódu, což umožňuje poměrně snadnou změnu požadovaného cílového kódu. Změna se bude týkat pouze zadní části překladače, vše ostatní může být zachováno,
zadní část provádí kontroly a optimalizace kódu a generuje cílový kód (popř. provádí interpretaci). Tato část je závislá na cílovém kódu a jakákoliv změna cílového kódu si vyžádá její úpravy.

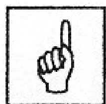
2.1 Analytická část překladače

Lexikální analyzátor



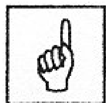
Lexikální analyzátor je první částí překladače, která se setká se zdrojovým programem. Lexikální analyzátor čte postupně znaky zdrojového programu a vytváří z nich lexikální symboly (tokeny) programu jako jsou identifikátory, čísla nebo klíčová slova. Pro každý lexikální symbol se uchovává jeho typ případně i atributy. Během analýzy vynechává lexikální analyzátor znaky, které pro program nemají význam (mezery, komentáře).

Syntaktický analyzátor



Syntaktický analyzátor dostává na vstup posloupnost symbolů programu, která je výsledkem práce lexikálního analyzátoru. Zjišťuje syntaktickou správnost zdrojového programu. Výstupem syntaktického analyzátoru je informace o syntaktické struktuře zdrojového programu, která bývá reprezentována např. derivačním stromem, nebo hlášení o chybě.

Sémantický analyzátor

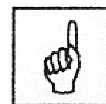


Sémantický analyzátor uzavírá analytickou část překladače. Vytváří vstup pro zpracování výstupu (generování cílového kódu, interpretace). V této časti překladu se provádějí především kontroly deklarací, typová kontrola apod. U operátorů se například kontroluje správný typ operandů, u proměnných se kontroluje, zda jsou deklarovány (pokud to konkrétní programovací jazyk vyžaduje) atd.

2.2 Syntetická část překladače

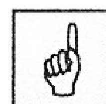
Generování cílového programu uzavírá celý překlad. Pokud program prošel bez chyb všemi výše uvedenými částmi, přichází na řadu optimalizování, které má za úkol zvýšit efektivitu vytvořeného programu. Pro tuto činnost je vhodnější převést program do tzv. intermediálního kódu. Ten se již blíží cílovému programu, případně jej lze v případě interpreta rovnou provádět. V dnešní době může být fáze optimalizování značně složitá, kromě optimalizací (vhodné datové typy apod.) se dnešní překladače potýkají s problémem různých instrukčních sad (MMX, 3DNow!, SSE, hyperthreading), a tak se optimalizování v různých překladačích liší různou mírou propracovanosti.

Intermediární kód



V případě kompilačního překladače je optimalizovaný intermediální kód převeden do cílového programu (např. strojový kód). V případě čistě interpretačního překladače je možné začít s interpretací ihned po ukončení sémantické analýzy, kdy je interpretu předána informace o požadovaných operacích. V dnešní době však využívají intermediálního kódu i překladače, které jinak pracují jako interpretační, což je důkaz toho, jak se některé rozdíly v obou přístupech ke strojovému překladu do jisté míry smazávají.

Cílový kód



Po dokončení analýzy zdrojového kódu generují překladače tzv. mezikód nebo též intermediární kód. Mezikód by se dal přirovnat ke strojovému kódu nějakého virtuálního počítače. V případě platforem Java a Microsoft.NET tomu tak skutečně je. Z důvodu platformové nezávislosti se vždy vytvoří pouze mezikód a ten je až v okamžiku spuštění překládán a interpretován. U platformy Java se mezikód nazývá Java Virtual Machine code (JVM) u platformy .NET pak Microsoft Intermediate Language (MSIL). Při konstrukci mezikódu již dochází ke konfrontaci jazyka vstupního a výstupního. Mezikód nemusí být nutně nějaký druh spustitelného kódu (i když v případě interpretu tomu tak je), ale pouze se může týkat transformace např. na tříadresový kód, který je charakteristický tím, že v instrukci mohou být maximálně tri operandy.

Vygenerovaný kód většinou není tak optimální, jako kdybychom celý program napsali například v jazyce assembleru. Díky snaze o zobecnění konstrukcí pro generování kódu tak dochází k nafukování kódu a k provádění zbytečných akcí. Zdrojem neoptimálnosti kódu ovšem z velké části bývá i sám programátor. K optimalizaci se dá přistupovat těmito způsoby:

1. Pokusit se upravit a zefektivnit kód, poté co je vytvořen. Tento přístup je také nazýván „pohled klíčovou dírkou". Hlavní myšlenkou je znalost výstupu překladače. Víme, které části se ve výstupu objevují zbytečně a podle toho také provádíme kroky optimalizace. Jednoduše se prochází výstupní kód, vyhledávají se kombinace a nahrazují se efektivnějšími. Velkou nevýhodou je hlavně množství takovýchto kombinací, které musí být zpracovány. Naopak výhodou dobře navržené optimalizace toho druhu,
je především jeho dokonalost. Všechny známé případy jsou odstraněny. Principem je tedy náhled, přes jakési okénko, nebo klíčovou dírku na část kódu. Samotná programová realizace, tak nemusí být částí překladače, ale překladač muže vyprodukovat kód, který pak převezme oddělený program a optimalizuje jej. Tento způsob se nazývá dvouprůchodový.

2. Pokusit se vytvářet lepší kód již v základu. Narozdíl od předchozí možnosti tento přístup již zasáhne celou strukturu překladače. Při každém kroku, je potřeba se rozhodnout, zda-li jednotlivé části kódu, které jsou produkovány, jsou nezbytně nutné. Příkladem může být identifikace konstant, identifikace nevyužitých proměnných a nedosažitelných částí kódu. V praxi zřejmě vidíme, že ideální je použití obou přístupů.

Optimalizace do značné míry zvyšuje složitost celého překladače a také prodlužuje dobu překladu. Právě proto je v moderních překladačích možnost optimalizace volitelná. Tedy předpokládáme-li častý překlad, jako například při vývoji aplikace, optimalizaci vypínáme, naopak při vytváření finální verze (release), je optimalizace zapnuta.

Jedním z kritérií pro výběr optimalizace je také to, zda chceme kód optimalizovat na rychlost, nebo velikost. Dohromady jsou oba přístupy

nesplnitelné. Jedním z příkladu optimalizace na rychlost jsou např. inline procedury v jazyce C++. Při překladu se zařadí kompletní algoritmus funkce, vždy na místo, kde je daná funkce volána. Dochází tak ke zvětšení kódu, ale pokud není funkce složitá a je často volána, pak změna velikosti je malou daní za výslednou efektivitu.

2.3 Tabulka symbolů

Tabulka symbolů

Tabulka symbolů je centrálním úložištěm dat všech částí překladače. Ve fázi analýzy slouží především k záznamu jednotlivých proměnných, jejich typu a výchozích hodnot a také k jejich jednoznačné identifikaci a zajištění jedinečnosti. V tabulce je také uložena informace o velikosti objektu, což může být později využito při dynamické či statické alokaci paměti. Při konstrukci tabulky a jejím dalším použití jsou často využívány metody podobné prohledávání databází jako tvorba indexu pro záznamy.

Typy implementace tabulky symbolů ovlivňuje potřeba dvou základních operací nad takovou tabulkou. Jedná se následující operace:

Vkládání – tato operace by měla umožnit vložit nový symbol do tabulky, přičemž by se samozřejmě měla provést kontrola, zda se již objekt stejného typu a názvu nevyskytuje. To může a nemusí znamenat chybu, např. deklarace forward v Pascalu umožňuje, aby se stejný identifikátor procedury vyskytl dvakrát.
Vyhledání – tato operace musí umět najít daný identifikátor a případně jeho atributy vrátit jako funkční hodnotu

Je jasné, že implementace bude značně ovlivňovat efektivitu těchto dvou operací. V zásadě můžeme rozlišit dva případy:

Jazyky bez blokové struktury programu (např. Basic, který neumožňuje práci s procedurami).

Jazyky s blokovou strukturou programu (např. Pascal, který umožňuje používat lokální proměnné, procedury).

Pro jazyky bez blokové struktury lze aplikovat následující postupy:

Neseřazené tabulky implementované jako pole nebo seznamy – tyto struktury jsou velmi jednoduché na implementaci, ovšem jak operace vyhledání, tak vkládání jsou neefektivní (mají obě lineární složitost).
Seřazené tabulky s binárním vyhledáváním – binární vyhledávání umožňuje snížit složitost vyhledávání na logaritmickou složitost, ovšem vkládání zůstane stejně složité.
Stromově strukturované tabulky – binární vyhledávací stromy umožňují zredukovat jak složitost vkládání, tak vyhledání, ovšem za cenu vyšší paměťové a implementační náročnosti.

Pro jazyky s blokovou strukturou je situace mnohem složitější, musíme si totiž zajistit lokální chování takové struktury. Pro tyto lokálně funkční struktury se většinou používá kombinace struktur z předchozího odstavce se strukturou typu zásobník. Ukládáme tedy do nich postupně symboly jako do zásobníku a při hledání symbolu s určitým jménem se vždy hledá nejbližší struktura od vrcholu zásobníku podle úrovně vnoření. Je tedy třeba rovněž mít jistou tabulku informací o volání vnořených funkčních bloků.

2.4 Intermediární kód

Již jsme se zmínili, že překladač většinou negeneruje přímo cílový kód, ale nejprve vytvoří nějaký mezikód – intermediární kód, který lze jednoduše optimalizovat. Existuje několik možností, jak takový kód může vypadat. Některé pouze zmíníme, ale jednou z nich se budeme hlouběji zabývat v rámci praktických implementací.

Grafová (stromová) reprezentace – například aritmetický výraz lze zapsat do formy syntaktického stromu, který lze pak průchodem typu „postorder“

vyhodnotit. To znamená, že vždy nejprve vyhodnotíme oba podstromy na hodnoty a teprve pak provedeme danou operaci.

Zásobníkový kód – vychází ze známé postfixové notace výrazu. V této notaci jsou vždy operátory až za svými operandy. Např. výraz a + b * c je v postfixové notaci zapsán jako a b c * + . Na základě této notace lze zapisovat instrukce zásobníkového kódu, které obsahují příkazy pro uložení proměnné na zásobník (VAR), přiřazení vrcholu zásobníku do proměnné (ASG) a instrukce běžných aritmetických operací (prováděné s operandy na vrcholu zásobníku a výsledek se uloží zpět do zásobníku). Kód pro náš případ by pak byl:

VAR a

VAR b

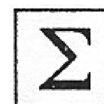
VAR c

MUL (násobení posledních dvou položek zásobníku)

ADD (sčítání posledních dvou položek zásobníku)

Tříadresový kód – spočívá v transformaci výrazů na sekvenci výrazů, které obsahují pouze nejvýše binární operace. Dále se v tomto kódu mohou využívat přiřazení, podmíněný a nepodmíněný skok, volání procedury se specifikací parametrů, indexované proměnné a také adresní operátory.

Nejdůležitější probrané pojmy:



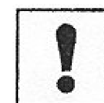
lexikální, syntaktická a sémantická analýza

mezikód, cílový kód

tabulka symbolů

intermediární kód

Úkoly k textu:



1. Vyjmenujte několik příkladů elementů, které kontrolujeme na úrovni lexikální, syntaktické a sémantické analýzy.

3 Lexikální analýza

Cíl:

Po prostudování této kapitoly pochopíte:

- pojem tokenu
- rozpoznávací sílu lexikální analýzy

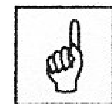
Průvodce studiem

Následující kapitola je věnována první fázi strojového překladu – lexikální analýze. Lexikální analyzátor má za úkol převést zdrojový program na posloupnost nejmenších částí s vlastním významem, které nazýváme symboly. Symboly dále slouží jako vstup pro syntaktický analyzátor.

3.1 Lexikální struktura jazyka

Pro popis symbolů vstupního jazyka používáme **regulární gramatiky**, případně **regulární výrazy.** I přes to, že regulárními výrazy lze popsat jen relativně jednoduché konstrukce, našly své uplatnění právě při lexikální analýze, neboť většina lexikálních konstrukcí běžných programovacích jazyků patří do třídy **regulárních jazyků**.

V překladači se pro jejich rozpoznávání používá **konečný automat**. Kromě samotného symbolu se do syntaktického analyzátoru posílají další související údaje, např. jakého druhu je daný symbol (operátor, klíčové slovo, identifikátor, atd.), u identifikátorů odkaz do tabulky identifikátorů (pro urychlení překladu, překladač poté nemusí pracovat s textem, ale pouze s číselným označením) apod. Celý soubor údajů předávaný z lexikálního do syntaktického analyzátoru označujeme anglickým termínem **token**.



Token

Regulární jazyk

Jazyk se nazývá regulární, pokud je vytvořen ze základních symbolů abecedy pouze s pomocí operací sjednocení, zřetězení a iterace. Formální definici lze najít například v [Ha03]. Jednoduchý regulární jazyk je například množina všech slov vyhovujících regulárnímu výrazu a*b*. Tento jazyk se bude skládat ze všech slov začínajících libovolným (i nulovým) počtem písmen „a“ a pokračujících libovolným (i nulovým) počtem písmen „b“.

Regulární výraz

Formálně je regulární výraz definován následujícím způsobem:

1. *a* je regulární výraz pro libovolný literál (znak abecedy, popř. prázdný symbol ε) *a*, popisující právě text *a*.
2. Pokud *A* a *B* jsou regulární výrazy, je *AB* regulární výraz, popisující zřetězení textů popsaných výrazy *A* a *B*.
3. Pokud *A* a *B* jsou regulární výrazy, je *A* + *B* regulární výraz, popisující buď text popsaný výrazem *A*, nebo text, popsaný výrazem *B*.
4. Pokud *A* je regulární výraz, pak *A** je regulární výraz, popisující libovolný počet opakování (včetně žádného opakování) textů popsaných výrazem *A*.
5. Pokud *A* je regulární výraz, je (*A*) regulární výraz popisující stejný jazyk. (Závorky slouží pouze pro vyjasnění priorit.)

Regulární výraz je tedy řetězec popisující určitou množinu řetězců.

Regulární gramatika

Gramatiku, kterou označujeme jako regulární, lze definovat takto:

gramatika G= (N, T, P, S) se nazývá regulární gramatika, jestliže každé pravidlo P je v jednom z tvarů $X \rightarrow wY$, $X \rightarrow w$, kde $X,Y \in N$, $w \in T^*$.

3.2 Rozpoznávání symbolů

Protože vstupem lexikálního analyzátoru je konstrukce ze třídy regulárních jazyků, budeme pro samotné rozpoznávání symbolů potřebovat konečný automat.

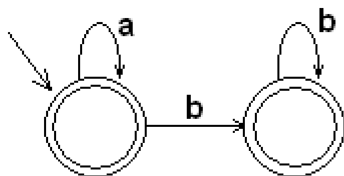
Konečný automat

Formálně jej lze definovat takto:

(Deterministickým) konečným automatem (DKA) nazýváme každou pětici $A = (Q, \Sigma, \delta, q_0, F)$, kde

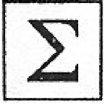
- Q je konečná neprázdná množina (množina stavů, stavový prostor),
- Σ konečná neprázdná množina (množina vstupních symbolů, vstupní abeceda),
- δ je zobrazen í $Q \times$
- $q_0 \in Q$ (počáteční stav, iniciální stav),
- $F \subseteq Q$ (množina koncových stavů, cílová množina).

Konečný automat pracuje tak, že provádí posloupnost přechodů. Přechod je určen stavem, ve kterém se automat nachází a symbolem, který je čten ze vstupního řetězce. Při přechodu přejde automat do nového stavu a přečte jeden vstupní symbol. Čtení vstupních symbolů se provádí zleva doprava počínaje nejlevějším symbolem vstupního řetězce. Konečný automat lze znázornit tzv. stavovým diagramem. Na následujícím obrázku je ukázán konečný automat rozpoznávající již dříve použitý jazyk vyhovující regulárnímu výrazu a*b*.



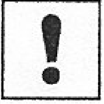
Stavový diagram automatu

Počáteční stav je označen v levém horním rohu šipkou a koncový stav je reprezentován dvojitým kruhem. V uvedeném případě jsou oba stavy koncové, aby bylo zaručeno rozpoznání opravdu všech slov vyhovujících regulárnímu výrazu a*b*.



Nejdůležitější probrané pojmy:

- lexikální analyzátor
- regulární jazyk
- konečný automat



Úkoly k textu:

1. Naprogramujte lexikální analyzátor pro reálné číslo Pascalovského stylu (tedy obsahuje celou část, může obsahovat desetinnou část a exponent, např. +123.456e-3). Použijte buď simulaci konečného automatu pro daný jazyk nebo vlastní řešení pomocí cyklů a podmínek.

4 Syntaktická analýza

Cíl:

Po prostudování této kapitoly pochopíte:

- činnost syntaktického analyzátoru
- vztah k bezkontextovým jazykům
- syntaktickou analýzu zdola nahoru a shora dolů
- metodu rekurzivního sestupu

Po prostudování této kapitoly se naučíte:

- vytvořit syntaktický analyzátor pro daný jazyk

Průvodce studiem

Syntaktický analyzátor (v angličtině označovaný jako parser) provádí samotnou analýzu vstupního jazyka. K tomu využívá posloupnost lexikálních symbolů (tokenů) získanou jako výsledek lexikální analýzy.

Úkolem syntaktického analyzátoru je rozpoznat, zda je program zapsán správným způsobem, např. zda na úvodní „begin“ bude navazovat někde koncové „end“ (za předpokladu, že to daný jazyk vyžaduje), ale také určuje, v jakém pořadí se budou provádět jednotlivé části příkazů, např. u „x + y*z“ rozpozná a určí, že nejdříve se bude násobit a pak až sčítat, nebo u vnořených příkazů zajistí, že nejdříve se vyhodnotí parametr funkce a teprve pak se daná funkce zavolá.

Úloha syntaktického analyzátoru je o něco složitější než úloha lexikálního analyzátoru, neboť narozdíl od regulárního jazyka lexikálních symbolů musí syntaktický analyzátor rozpoznávat složitější **bezkontextový jazyk**. Během své práce vytváří syntaktický analyzátor tzv. **derivační strom** (anglicky *parse*

tree), který popisuje strukturu vstupního programu. Podle toho, jak je konstruován derivační strom, rozlišujeme dvě základní metody syntaktické analýzy:

1. metoda **shora dolů** vytváří derivační strom od kořene k listům,
2. metoda **zdola nahoru** vytváří derivační strom od listů ke kořeni.

4.1 Základní pojmy

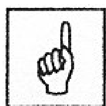
Bezkontextový jazyk

Jedná se o obecnější třídu jazyků než byly regulární jazyky popsané v kapitole předchozí. Využitím bezkontextových gramatik získáme tedy silnější popisné prostředky. Bezkontextové gramatiky jsou nejvhodnější pro popis současných programovacích jazyků. Teoretickým modelem pro analýzu bezkontextových jazyků je **zásobníkový automat**.

Zásobníkový automat

Je sedmice $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, kde

- Q je konečná neprázdná množina (množina stavů, stavový prostor),
- Σ konečná neprázdná množina (množina vstupních symbolů, vstupní abeceda),
- Γ je konečná abeceda zásobníkových symbolů
- δ je zobrazení $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$ (přechodová funkce),
- $q_0 \in Q$ je počáteční stav,
- $Z_0 \in \Gamma$ je symbol, který je na počátku uložen na zásobníku,
- $F \subseteq Q$ je množina koncových stavů.

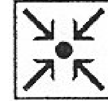


Derivační strom

Derivační strom

Tento pojem si nejlépe vysvětlíme na následujícím příkladu jednoduché bezkontextové gramatiky.

Řešený příklad 1:



G = ({S}, {a, +, *}, P, S)

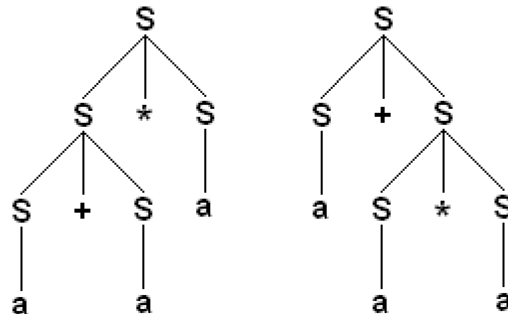
P: S → S+S | S*S | a

Nyní provedeme odvození (derivaci) věty a+a*a:

S → S*S → S+S*S → a+S*S → a+a*S → a+a*a, nicméně ke stejné větě můžeme dojít i takto:

S → S+S → S+S*S → S+S*a → S+a*a → a+a*a.

V každém kroku odvozování jsme přepsali jeden neterminální symbol. V prvním případě jsme však vždy přepisovali neterminální symbol, který byl nejvíce vlevo, takové odvozování nazýváme levá derivace. Analogicky, pokud přepisujeme vždy neterminální symbol, který je nejvíce vpravo, pak takové odvození označujeme jako pravou derivaci. Na obrázku vidíme derivační stromy pro výše uvedená odvození.



Derivační stromy

Pokud lze dojít k alespoň jedné větě pomocí více odvození (existuje více různých derivačních stromů pro tutéž větu), pak takovou gramatiku nazýváme nejednoznačnou. Gramatika, která má pro každou větu právě jeden derivační strom, se nazývá jednoznačná.

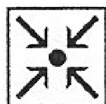
4.2 Syntaktická analýza shora dolů

Tato metoda syntaktické analýzy vychází ze startovacího symbolu gramatiky a postupuje k terminálním symbolům, které tvoří analyzovanou větu. Derivační strom vytváříme podle levé derivace od kořene k listům. Syntaktickou analýzu metodou shora dolů také nazýváme procesem hledání levého rozkladu analyzované věty, což je posloupnost čísel pravidel použitých při levé derivaci.

Řešený příklad 2:

Pro následující příklad vytvoříme gramatiku, která generuje matematické výrazy v infixovém tvaru:

G = ({S, T, U}, {a, b, c, +, -, *, /}, P, S)



Pravidla:	Čísla pravidel:
S → T+S | T-S | T	1 | 2 | 3
T → U*T | U/T | U	4 | 5 | 6
U → (S) | a | b | c	7 | 8 | 9 | 10

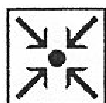
Nyní provedeme levou derivaci věty (a+b)*c:

S → T → U*T → (S)*T → (T+S)*T → (U+S)*T → (a+S)*T → (a+T)*T → (a+U)*T → (a+b)*T → (a+b)*U → (a+b)*c

Postupně jsme použili pravidla 3, 4, 7, 1, 6, 8, 3, 6, 9, 6, 10 a tato posloupnost tvoří v tomto případě levý rozklad věty (a+b)*c v gramatice G.

4.3 Syntaktická analýza zdola nahoru

V tomto případě postupujeme při konstrukci derivačního stromu od listů (terminální symboly, které tvoří analyzovanou větu) pomocí pravých derivací ke kořeni stromu (je ohodnocen výchozím neterminálním symbolem gramatiky). Syntaktickou analýzu metodou zdola nahoru pak nazýváme procesem hledání pravého rozkladu analyzované věty, což je posloupnost čísel pravidel použitých při pravé derivaci v obráceném pořadí.



Řešený příklad 3:

Význam obráceného pořadí si nejlépe ukážeme na příkladě.

G = ({S, T, U}, {a, b, c, +, -, *, /}, P, S)

Syntaktická analýza

Pravidla: Čísla pravidel:

S → T+S | T-S | T 1 | 2 | 3

T → U*T | U/T | U 4 | 5 | 6

U → (S) | a | b | c 7 | 8 | 9 | 10

Nyní provedeme pravou derivaci věty (a+b)*c:

S → T → U*T → U*U → U*c → (S)*c → (T+S)*c → (T+T)*c → (T+U)*c → (T+b)*c → (U+b)*c → (a+b)*c

Postupně jsme použili pravidla 3, 4, 6, 10, 7, 1, 3, 6, 9, 6, 8.

Nyní metodou zdola nahoru analyzujeme tutéž větu (hledáme pravý rozklad):

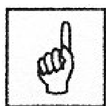
(a+b)*c → (U+b)*c → (T+b)*c → (T+U)*c → (T+T)*c → (T+S)*c → (S)*c → U*c → U*U → U*T → T → S

Postupně jsme použili pravidla 8, 6, 9, 6, 3, 1, 7, 10, 6, 4, 3 a vidíme, že pravý rozklad je opravdu obrácená posloupnost pravidel použitých při pravé derivaci.

Při syntaktické analýze se můžeme setkat s problémem výběru vhodného přepisovacího pravidla (viz. předchozí příklad). Při analýze shora dolů je to situace, kdy máme pro jeden neterminální symbol více pravidel, jejíchž pravé strany začínají stejným podřetězcem. U metody zdola nahoru může nastat případ, kdy máme pro aktuálně přepisovaný řetězec více pravidel s odpovídající pravou stranou a tedy nevíme, kterou levou stranu použít. Obecně lze řešit problém výběru vhodného pravidla dvěma způsoby:

- analýza s návratem – je založena na případném návratu o krok zpět (po výběru nevhodného pravidla) a aplikaci jiného pravidla,
- deterministická analýza – je založena na dodatečných informacích o analyzovaném řetězci.

Analýza s návratem se v praxi nevyužívá, neboť většina konstrukcí běžných programovacích jazyků umožňuje deterministickou analýzu bez návratů. Pro deterministickou analýzu shora dolů využíváme tzv. LL (k) gramatiky, kde k označuje počet symbolů vstupního řetězce, které potřebujeme znát „dopředu“. Pro syntaktickou analýzu zdola nahoru pak takové gramatiky označujeme jako LR (k). Speciálním případem LR (1) gramatik jsou tzv. LALR (1) gramatiky, s nimiž pracují moderní generátory překladačů.



4.4 Backusova-Naurova forma

Backusova-Naurova

Dalším přehledným a hlavně v praxi ještě více využívaným způsobem zápisu syntaxe jazyka je takzvaná **Backusova-Naurova forma (BNF)**. Jde o zápis podobný bezkontextové gramatice, ale přitom bližší spíše programátorům, resp. praxi.

BNF obsahuje podobně jako BKG neterminály, které se uvádějí do úhlových závorek a přepisují skrze symbol := na řetězce terminálních a neterminálních symbolů. Jde tedy o pravidla tvaru:

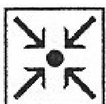
$<X> := \alpha_1 \mid \ldots \mid \alpha_n$

Pro přehlednější zápis je však ještě lepší modifikace BNF zvaná EBNF (Extended BNF) – rozšířená BNF, která zjednodušuje zápis opakovaně používaných, příp. podmíněně vyskytujících se výrazů. Umožňuje následující zápisy:

$\{\alpha\}$ – znamená, že výraz se vyskytuje v libovolném počtu (ekvivalent operace iterace)

$\{\alpha\}_n^m$ - znamená, že výraz se vyskytuje v počtu nejméně n a nejvýše m (ekvivalent operace mocniny od n do m)

$[\alpha]$ – znamená, že výraz se může a nemusí na daném místě vyskytnout - je to ekvivalentní zápisu $\{\alpha\}_0^1$



Gramatika z předchozího příkladu by v BNF mohla být zapsána například takto:

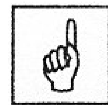
<aritmetický výraz+> := <aritmetický výraz*>{+<aritmetický výraz*>}

<aritmetický výraz*> := <operand/podvýraz>{*<operand/podvýraz>}
<operand/podvýraz> := (<aritmetický výraz+>) | x

BNF umožňuje přehledný zápis a navíc i jednoduchý přechod k některým typům SA. S pomocí BNF je zapsána například celá gramatika jazyka Pascal v učebnici [Ji88]. Příkladem může být deklarace podmíněného příkazu:

<podmíněný příkaz> := if <booleovský výraz> then <příkaz>|
if <booleovský výraz> then <příkaz> else <příkaz>

4.5 Metoda rekurzivního sestupu

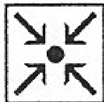


Rekurzivní sestup

Velice oblíbenou, jednoduchou a přehlednou metodou vedoucí přímo ke zdrojovému kódu je **metoda rekurzivního sestupu** (**Recursive Descent Parsing**) [Le02]. Metoda je založena na principu analýzy „shora dolů“ a je tedy blízká LL gramatikách.

Metoda rekurzivního sestupu spočívá v konstrukci procedur strukturovaného programovacího jazyka přesně dle Backusovy-Naurovy formy (BNF), kde je každému neterminálu přiřazena jedna procedura a je volána procedura GetChar (načítající vždy následující symbol slova) před každým terminálem. Výskyt neterminálu v pravidle je v proceduře nahrazen rekurzivně voláním příslušné procedury. Iterace a podmínka v Backus-Naurově formě je nahrazena jednoduše jejich programátorskými protějšky. Vyžaduje se, aby jazyk byl definován LL(k) gramatikou. Z hlediska časové složitosti jde opět o obecně neefektivní metodu s exponenciální časovou složitostí, nicméně pro jednoduché gramatiky z praxe je použitelná a zejména je její výhodnou vysoká čitelnost kódu ve vztahu k výchozí gramatice. Navíc existuje modifikace tzv. **packrat parser**, která pro omezenou třídu takzvaných **parsing expression grammars** pracuje v lineární čase. Také je tento postup flexibilní, protože umožňuje kdykoliv změnit a přidat syntaktický element bez nutnosti měnit celý kód, ale pouze dotčenou část gramatiky (například změna struktury číslo z celého čísla na reálné znamená pouze změnu procedury reprezentující tento

element). Podívejme se nyní na příklad gramatiky pro generování aritmetických výrazů se sčítáním, násobením, číslicemi a vnořenými závorkovanými strukturami.



Gramatiku v Backusově-Naurově formě pro náš zjednodušený příklad (pouze s číslicemi místo identifikátorů) lze zapsat takto:

<Výraz> ::= <Term> { + <Term>}
<Term> ::= <Faktor>{ * <Faktor>}
<Faktor> ::= 0|1|2|…|9|(<Výraz>)

Nyní se schématicky pokusíme ukázat (nejde o zcela hotový kód), jak bychom sestrojili SA metodou rekurzivního sestupu pro tuto gramatiku v jazyce Pascal. Tento kód, pak umožňuje nejen SA, ale i detekci možných chyb. Nejprve sestrojíme proceduru, která zapouzdřuje celou činnost SA. Její hlavička může vypadat například takto:

```
procedure SyntaktickaAnalyza(infix:string;var err,pos:word);
        {procedura analyzuje aritmetický výraz infix, err obsahuje číslo chyby, pos
obsahuje pozici ,kde analýza skončila}
```

Používají se proměnné infixpos (pozice aktuálně čteného znaku ze vstupu), ch (aktuální znak).

Analyzátor dále obsahuje nezbytný lexikální analyzátor pro načítaní jednotlivých symbolů (v našem zjednodušeném případě jde o jednoznakové symboly). Lexikální analýza je realizována procedurou GetChar, která ukládá znak do proměnné ch a případně provede detekci chybové situace err=2, pokud načteme zcela nepřípustný znak.

```
procedure Getchar;       {čte znak z infixu do proměnné ch}
begin
  if  err=0 then
```

```
    begin
      Inc(infixpos);
      if infixpos<=Length(infix) then ch:=infix[infixpos] else ch:=#0;
      ch:=Upcase(ch);
      if not((ch in cislice)or(ch in ['(',')','*','+'])) then err:=2;   {ošetření nežádoucích znaků}
    end;
end;
```

Jádrem analyzátoru jsou jednotlivé rekurzivní procedury Výraz (sčítaní), Term (násobení), Faktor (číslice, vnořený závorkovaný výraz). Výraz přesně podle BNF buď volá podřízený Faktor nebo čte terminální symboly.

```
procedure Vyraz;              {výraz s nižší prioritou}
begin
  if err=0 then
    begin
      Term;
      while (ch='+') do
        begin
            Getchar;                  {sčítání}
            Term;
        end;
    end;
end;

procedure Term;               {výraz s vyšší prioritou}
begin
  if err=0 then
    begin
      Faktor;
      while (ch='*') do
        begin
            Getchar;                  {násobení}
            Faktor;
        end;
    end;
end;
```

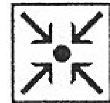
A dále musíme sestrojit proceduru pro Faktor, která bude mít vzhledem k jinému charakteru přepisovaného řetězce i jiný kód.

```
procedure Faktor;   {synt. analýza operandu}
begin
 if err=0 then
   begin
    case ch of
    '0'..'9':
       begin                       {anal. číslic}
        Getchar;
       end;
    '(':begin
        Getchar;
                        {analýza výrazu se závorkou}
        Vyraz;
        if (ch<>')')and(err=0) then err:=4   {chyba- není ukončen závorkou}
        else if err=0 then
          begin
            Getchar;
          end;
       end;
    else if err=0 then err:=5;    {nebyl detekován ani vyraz,ani číslice}
    end;
   end;
```

Faktor tedy rozlišuje dvě situace – buď jde o číslici nebo jde o výraz začínající závorkou a ukončený opačnou závorkou. Logicky tedy můžeme odhalit další dvě chyby (err=4, když chybí závorka, err=5, když není detekován ani výraz ani číslice).

Pozn. Samozřejmě, že chybové detekce by mohly odhalit ještě další problematické konstrukce – např. skončení nejvyššího volání procedury Výraz před přečtením posledního znaku apod.

Řešený příklad 4:

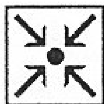


Ilustrujme nyní průběh výpočtu procedury Výraz na výrazu 5 + 3 * 2.

Infixová notace	**Aktuální znak**	**Aktuální procedura**	**Návrat do procedury**
5 + 3 * 2	5	Výraz	
5 + 3 * 2	5	Term	
+ 3 * 2	+	Faktor	Term
+ 3 * 2	+	Term	Výraz
3 * 2	3	Výraz (+)	
3 * 2	3	Term	
* 2	*	Faktor	Term
2	2	Term (*)	
		Faktor	Term (*)
		Term (*)	Výraz (+)
		Vyraz (+)	

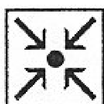
4.6 Přímá implementace analytické části překladače

Rozeberme nyní jádro vyhodnocení výrazu v infixní formě. Toto jádro obsahuje především základní proceduru Compile, která provádí kompilaci aritmetického v infixové (tedy přirozené) notaci do notace postfixové. Ta je vhodná pro zpracování pomocí počítače např. vyhodnocení pomocí zásobníku.



Aritmetický výraz v infixové (přirozené) notaci 5 + 3 * 2 lze pomocí této procedury přeložit na výraz v postfixové notaci 5 3 2 * +. Ta je konstruována tak, že místo tvaru, kde je operátor mezi operandy, má operátor až za oběma operandy.

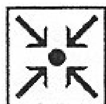
Tento výraz lze pak pomocí zásobníku vyhodnotit tak, že čteme jednotlivé symboly a provádíme s nimi tyto dvě operace:
Je-li čtený symbol operandem, pak ulož operand na zásobník.
Je-li čtený symbol operátorem, pak vyber ze zásobníku posledních n operandů (kde n je arita operátoru; např. pro + je n = 2). Proveď operaci dle operátoru s vybranými operandy a výsledek ulož na zásobník.



Pro výraz 5 + 3 * 2 vezměme jeho postfixovou notaci 5 3 2 * + a vyhodnoťme jej s pomocí zásobníku.

Nepřečtená část	**Aktuální znak**	**Zásobník**	**Vybírané symboly**	**Operace**
5 3 2 * +	5			
3 2 * +	3	5		
2 * +	2	3 5		
* +	*	2 3 5	2 3	2 * 3 = 6
+	+	6 5	6 5	6 + 5 = 11
		11		

Obsah zásobníku po přečtení slova je roven hodnotě výrazu. Postup by samozřejmě bylo možno zobecnit na složitější čísla nebo proměnné, ale vyžadovalo by to složitější struktury zásobníku.



Syntaktická analýza

Máme-li postfixový tvar výrazu je jednoduché provést jeho vyhodnocení, ale mnohem složitější je samotná kompilace. Procedura Compile při svém běhu používá procedury sestrojené již zmiňovanou metodou rekurzivního sestupu.

Gramatiku v Backusově-Naurově formě pro náš zjednodušený příklad (pouze s číslicemi místo identifikátorů) lze zapsat takto:

<Výraz> ::= <Term> { <+ -> <Term>}; <+ -> ::= + | -
<Term> ::= <Faktor> { <* /> <Faktor>}; <* /> ::= * | /
<Faktor> ::= 0|1|2|…|9

Pozn. Zavedené neterminály <+ -> a <* /> jsou uvedeny pouze z důvodu dodržení notace BNF. Jsou zjevně tak jednoduché, že pro ně není nutné tvořit samostatné procedury při použití metody rekurzivního sestupu.

```
procedure Compile(infix1:string;var postfix1:string;var err1,pos1:word);
        {procedura kompilace výrazu -infix1 obsahuje vstupní vyraz
                                    -postfix1 obsahuje přeložený výraz
                                    -err1 obsahuje číslo chyby (viz demopgm)
                                    -pos1 obsahuje pozici ,kde kompilace skončila}
```

Modul dále obsahuje nezbytný lexikální analyzátor pro načítaní jednotlivých symbolů (v našem zjednodušeném případě jde o jednoznakové symboly). Lexikální analýza je realizována procedurou GetChar.

```
procedure Getchar;        {čte znak z infixu do proměnné ch}
begin
  if  err=0 then
    begin
      Inc(infixpos);
      if infixpos<=Length(infix2) then ch:=infix2[infixpos] else ch:=#0;
      ch:=Upcase(ch);
      if not((ch in pismeno)or(ch in cislice)or(ch in ['(',')','-','*','/','+'])
```

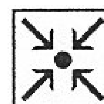
```
    or(ch in ignore)) then err:=2;   {ošetření nežádoucích znaků}
  end;
end;
```

Jádrem modulu jsou jednotlivé rekurzivní procedury Výraz (sčítaní, odčítání), Term (násobení, dělení), Faktor (identifikátor, unární plus a mínus, vnořený závorkovaný výraz), Identifikátor. Pro příklad uveďme nejvyšší proceduru Výraz.

```
procedure Vyraz;            {výraz s nižší prioritou}
begin
 if err=0 then
  begin
   Term;
   while (ch='+')or(ch='-') do
    begin
     case ch of
     '+': begin
            Getchar;
            Checkignore;     {sčítání}
            Term;
            postfix2:=postfix2+'+';
          end;
     '-': begin
            Getchar;
            Checkignore;     {odčítání}
            Term;
            postfix2:=postfix2+'-';
          end;
     end;
    end;
  end;
end;
```

Podobně jako u logických výrazu v aplikaci GERDS se zde volá procedura nižší v hierarchii (Term) a toto volání se opakuje podle počtu výskytu symbolu operace sčítání nebo odčítání. Důležité pro kompilaci do postfixové notace je implicitní zapamatování načteného symbolu operace (+, -) a to tím, že program je vždy v příslušné větvi (+, -) a po návratu z volané procedury Term se do proměnné postfix přidá příslušný operátor.

Ilustrujme to opět na výrazu 5 + 3 * 2 (pro jednoduchost jsou identifikátory čísla již v proceduře Faktor).

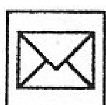


Infixová notace	**Aktuální znak**	**Postfixová notace**	**Aktuální procedura**	**Návrat do procedury**
5 + 3 * 2	5		Výraz	
5 + 3 * 2	5		Term	
+ 3 * 2	+	5	Faktor	Term
+ 3 * 2	+	5	Term	Výraz
3 * 2	3	5	Výraz (+)	
3 * 2	3	5	Term	
* 2	*	5 3	Faktor	Term
2	2	5 3	Term (*)	
		5 3 2	Faktor	Term (*)
		5 3 2 *	Term (*)	Výraz (+)
		5 3 2 * +	Vyraz (+)	

Tyto principy lze v zásadě použít i pro mnohem složitější jazyky než jsou obyčejné aritmetické výrazy. Princip rekurzivního vnořování lze nalézt kupříkladu u blokově orientovaných jazyků.

Nejdůležitější probrané pojmy:

- syntaktická analýza
- bezkontextový jazyk
- syntaktická analýza shora dolů a zdola nahoru
- Backusova-Naurova forma
- metoda rekurzivního sestupu
- postfixová notace



Korespondenční úkol:

Sestrojte BNF pro jazyk aritmetických výrazů, kde můžete používat operace sčítání, odčítání, násobení a dělení a dále operandy – reálná čísla Pascalovského stylu (tedy obsahuje celou část, může obsahovat desetinnou část a exponent, např. +123.456e-3) a rovněž můžete do závorek vnořit výraz stejného typu. Lexikální analyzátor může buď vracet tokeny – pro reálné číslo je pak nutné sestrojit složitější analýzu nebo lze tento element zařadit přímo do gramatiky a analyzovat syntaktickým analyzátorem (v tom případě lexikální analyzátor zůstane triviální).

Pro sestrojenou gramatiku vytvořte metodou rekurzivního sestupu program v libovolném strukturovaném programovacím jazyce (např. Pascal), který bude provádět syntaktickou analýzu libovolného výrazu a jeho chybovou detekci. Dále tento program musí přeložit výraz do postfixové notace a vyhodnotit ho zásobníkem na příslušnou číselnou hodnotu výrazu.

5 Automatizované nástroje

Cíl:

Po prostudování této kapitoly pochopíte:

- význam automatizovaných nástrojů pro tvorbu lexikálních a syntaktických analyzátorů

Průvodce studiem

Napsat překladač lze mnoha způsoby. Přímým naprogramováním buď v samotném assembleru nebo v některém z vyšších programovacích jazyků. Nicméně existují také speciální programové nástroje, které nám dokáží s vytvářením překladače velmi pomoci. Některé z nich patří do skupiny úzce specializovaných nástrojů, např. konstruktory lexikálních a syntaktických analyzátorů, jiné jsou komplexní systémy na vytváření celých překladačů..

V následující kapitole si představíme některé automatizované nástroje, které pomáhají programátorům při konstruování překladačů, či dokonce zvládají celý proces generování překladačů pouze s minimálním zásahem programátora. Představení je velmi stručné a má za cíl ukázat, že nástrojů pro generování překladačů je velké množství a tedy si každý zájemce o tuto problematiku jistě najde vhodný nástroj právě pro svou práci.

LEX

Generátor lexikálních analyzátorů. Vstupem je soubor regulárních výrazů a kódu, který se má vykonat při nalezení řetězce odpovídajícího některému z regulárních výrazů. Pro rozpoznávání generuje deterministický konečný automat. Výsledný lexikální analyzátor je generován v jazyce C nebo

jazyce Ratfor (jazyk vycházející z Fortranu). Je dostupný pro operační systémy UNIX, GCOS a OS/370.

FLEX

Tento program vychází z velmi dobře známého generátoru lexikálních analyzátorů LEX. Jedná se o software vydaný pod licencí GNU/GPL a je tedy dostupný zdarma i se zdrojovými kódy. Generuje lexikální analyzátor v jazyce C nebo C++ a spolupracuje s generátory syntaktických analyzátorů typu YACC.

YACC

Jedná se generátor syntaktických analyzátorů dostupný pro operační systém Unix navržený firmou AT&T. Generuje tzv. parser – část překladače, která má za úkol provádět syntaktickou analýzu vstupního textu. Vstupem je LALR (1) gramatika v Backus-Naur formě, výstupem pak analyzátor v jazyce C. Na jeho základě později vzniklo velké množství nejrůznějších generátorů syntaktických analyzátorů.

ACCENT

Další z generátorů syntaktických analyzátorů. Můžeme s ním pracovat stejně jako s programem Yacc, nicméně na rozdíl od něj neklade žádná omezení na typ vstupní gramatiky. Zápis vstupní gramatiky se tím zjednodušuje. Daní za možnost obecnějšího vstupu jsou zvýšené nároky vygenerovaných analyzátorů na systémové prostředky počítače.

ANAGRAM

Multifunkční generátor LALR (1) syntaktických analyzátorů v jazyce C / C++. Obsahuje několik utilit, díky nimž nabízí interaktivní způsob práce. Například modul File Trace umožňuje testování vytvářené gramatiky na pokusném vstupním souboru bez samotného kompilování analyzátoru. Anagram je určen pro platformu Win32 a existuje pouze ve formě plné placené verze nebo časově omezené zkušební verze (funkčnost 30 dní od data instalace).

Automatizované nástroje

HAPPY

Generátor syntaktických analyzátorů, jehož výstup je v jazyce Haskell. Šířen je zdarma včetně zdrojových kódů, případně ve formě binárních souborů pro platformy Linux a Windows.

BISON

Víceúčelový generátor syntaktických analyzátorů. Vstupem je LALR (1) bezkontextová gramatika a výstup je zapsán v jazyce C. Bison je shora kompatibilní s generátorem Yacc a použitelná je tedy každá správně zapsaná gramatika pro Yacc. Bison je dostupný pro operační systémy Unix i Windows.

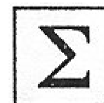
ELI

Komplexní systém pro generování překladačů. Zahrnuje nástroje pro řešení většiny problémů, které souvisí s implementací jazyka. Všechny součásti systému Eli jsou vzájemně propojeny a je zaručena vzájemná kompatibilita rozhraní jednotlivých modulů. Výstupním jazykem je jazyk C. Šířen je zdarma a existují verze pro operační systémy Windows, POSIX, OSX a další.

GENTLE

Univerzální sada nástrojů pro generování překladačů. Pokrývá celou oblast od analýzy až po syntézu. Pro osobní použití a vzdělávací účely je k dispozici zdarma. Jeho komerční verze jsou nasazovány ve strojírenském průmyslu.

Nejdůležitější probrané pojmy:



- automatizované nástroje pro lexikální analýzu
- automatizované nástroje pro syntaktickou analýzu

6 FLEX

Cíl:

Po prostudování této kapitoly pochopíte:

- činnost programu FLEX
- tvorbu definic a akcí pro FLEX

Po prostudování této kapitoly se naučíte:

- používat program FLEX pro tvorbu lexikálních analyzátorů

Průvodce studiem

V následující kapitole si objasníme způsob práce s jedním z nejznámějších generátorů lexikálních analyzátorů. Konkrétně se jedná o program Flex v poslední verzi 2.5.4a. Stejně jako ostatní nástroje pro výstavbu překladačů byl i program Flex vytvořen primárně pro prostředí operačních systémů typu Unix, nicméně pro účely tohoto textu budeme pracovat s verzí pro operační systém Windows.

V následující kapitole si objasníme způsob práce s jedním z nejznámějších generátorů lexikálních analyzátorů. Konkrétně se jedná o program Flex v poslední verzi 2.5.4a. Stejně jako ostatní nástroje pro výstavbu překladačů byl i program Flex vytvořen primárně pro prostředí operačních systémů typu Unix, nicméně pro účely tohoto textu budeme pracovat s verzí pro operační systém Windows.

6.1 Instalace

Naším prvním krokem bude samotné získání programu Flex pro Windows. Ten lze získat například na domovské stránce programu http://www.gnu.org/software/flex/ ve formě instalačního souboru o velikosti 1,2 MB. Samotná instalace je přímočará a rychlá, adresář s nainstalovaným programem zabírá příjemných 2,5 MB.

Instalace vytvoří v nabídce Start programovou skupinu Flex, která však již na první pohled postrádá důležitý prvek. Tím je samotný spouštěcí soubor aplikace. Ten najdeme v adresáři s nainstalovaným programem. Standardně c:\Program Fines\GnuWin32\bin. Zda najdeme dva spustitelné soubory flex.exe a flex++.exe. Flex totiž oproti svému předchůdci Lex přidává možnost generování zdrojového kódu v objektově orientovaném jazyce C++. My budeme používat první z nich, který generuje lexikální analyzátor v jazyce C. Druhou možností je použití varianty flex++, která předává na výstup zdrojový program v jazyce C++. Základní schéma práce generátoru Flex je znázorněno na následujícím obrázku:

Flex - schéma práce

Vstupem programu Flex je textový soubor, který obsahuje popis generovaného lexikálního analyzátoru. Protože se jedná o čistě textový soubor, není třeba dodržet standardně uváděnou příponu .l. Tento soubor se připojuje jako parametr při spouštění flex.exe. Výstupem je pak soubor v jazyce C standardně pojmenovaný lex.yy.c, který obsahuje hlavní funkci yylex(). Tento zdrojový kód je následně třeba přeložit některým překladačem jazyka C, čímž získáme spustitelnou aplikaci pro lexikální analýzu.

6.2 Volitelné parametry

Při spouštění programu Flex máme k dispozici celou řadu volitelných parametrů, z nichž ty nejdůležitější jsou uvedeny v následujícím seznamu:

-i	-výstupem bude lexikální analyzátor, který nerozlišuje velká a malá písmena ve vstupním souboru. Velikost písmen na vstupu je pro porovnávání ignorována, nicméně rozpoznaný text předávaný do proměnné *yytext* zůstane beze změny (původní velikosti písmen zůstávají zachovány).
-l	-zapíná režim maximální kompatibility s původní AT&T implementací Lex. Tato volba se projeví snížením výkonu.
-p	-generuje zprávu na *stderr* o výkonu vytvářeného lexikálního analyzátoru. Zpráva obsahuje komentáře k částem definičního souboru, které způsobují vážné ztráty výkonu (použití trailing apod.).
-s	-potlačuje implicitní pravidlo (vstup, který neodpovídá žádnému pravidlu, je poslán na standardní výstup). V případě nalezení vstupu, který neodpovídá žádnému pravidlu, se analyzátor předčasně ukončí s chybou.
-t	-přesměruje vytvořený analyzátor na standardní výstup místo do souboru lex.yy.c.
-w	-potlačuje výstražné zprávy.
-V	-předá verzi programu Flex na standardní výstup.
-+	-výsledný analyzátor bude vytvořen v jazyce C++.
-P*předpona*	-mění implicitní předponu yy všem globálním proměnným a funkcím na *předpona*. Navíc také mění jméno výstupního souboru z implicitního lex.yy.c na lex.*předpona*.c. Tato volba umožňuje použití více analyzátorů v jedné aplikaci.

6.3 Formát vstupního souboru

V následující části si popíšeme strukturu vstupního souboru. Obecný formát vypadá následovně:

```
Definice
```

```
%%
Pravidla
%%
Uživatelský kód v jazyce C
```

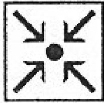
Vstupní soubor pro Flex se skládá ze tří částí, které jsou od sebe oddělené řádky obsahujícími pouze dva znaky pro procento těsně za sebou. První část nám slouží především pro deklarace jmen, které nám později usnadní psaní pravidel. Také zde definujeme tzv. startovní podmínky a zároveň je možné definovat zde proměnné, které poté využijeme ve svém kódu.
Druhá část obsahuje pravidla lexikálního analyzátoru. Každé pravidlo je tvořeno dvojicí regulární výraz a kód v jazyce C, který je vykonán, je-li daný regulární výraz ve vstupním souboru rozpoznán.
Třetí část obsahuje vlastní programátorův kód a je beze změny okopírována do výstupního zdrojového programu. V této sekci si tedy můžeme definovat vlastní pomocné funkce případně vlastní návrh funkce *main ()*. Tuto část lze také zcela vynechat. Funkce *main ()* bude poté vygenerována automaticky v následujícím tvaru:

```
main( argc, argv )
int argc;
char **argv;
  {
   ++argv, --argc;  /* skip over program name */
   if ( argc > 0 )
          yyin = fopen( argv[0], "r" );
   else
          yyin = stdin;

   yylex();
  }
```

Analyzátor zpracovává vstupní soubor *yyin*, který je implicitně nastaven na standardní vstup *stdin*. Takto implicitně definovaná funkce *main ()* se pokusí pro analýzu otevřít soubor uvedený jako parametr výsledného lexikálního analyzátoru a pokud spustíme vytvořený analyzátor bez parametru, bude očekávat analyzovaný text na standardním vstupu. Samotnou analýzu provádí hlavní funkce *yylex ()*.
Základní představení máme za sebou a proto si ukážeme první příklad. Vytvoříme si textový soubor, u kterého budeme předpokládat, že obsahuje pouze čísla a slova. Čísla budou celá a za slovo budeme považovat libovolnou

kombinaci písmen i samostatná písmena. Tento soubor předáme analyzátoru, který nám na výstupu sdělí, kolikrát rozpoznal ve vstupním souboru nějaké slovo a kolikrát narazil na číslo. Vstupní soubor pro Flex bude vypadat následovně:



Řešený příklad 5:

```
    int pocet_slov = 0, pocet_cisel = 0;
CISLO [0-9]+
%option noyywrap
%%
{CISLO}                 ++pocet_cisel;
[a-z][a-z]*        ++pocet_slov;
%%
int main(int argc, char **argv)
{
  ++argv, --argc;  /* skip over program name */
   if ( argc > 0 )
         yyin = fopen( argv[0], "r" );
   else
         yyin = stdin;
yylex ();
printf ("Pocet slov: %d, pocet cisel: %d\n",pocet_slov, pocet_cisel);
getch();
return 0;
}
```

Pokud působí výše uvedený příklad složitě, nic se neděje. Všechny důležité věci si řekneme později. Zařadili jsem zde tento příklad, abychom si na něm mohli vyzkoušet celý proces generování a následné kompilace našeho prvního lexikálního analyzátoru.

Celý příklad si uložíme do textového souboru první.l a z příkazové řádky spustíme program Flex, kterému dáme tento soubor jako parametr.

flex první.l

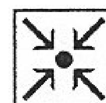
Výsledkem bude soubor lex.yy.c, což je zdrojový soubor našeho analyzátoru. Ten předáme některému z překladačů jazyka C, který nám z něj vytvoří spustitelnou aplikaci. Podívejme se ještě jednou na soubor s popisem analyzátoru a v definiční sekci najděme řádek, který obsahuje *%option noyywrap*. Pokud tento řádek vynecháme, následně vygenerujeme soubor lex.yy.c a ten se pokusíme zkompilovat, překlad skončí chybou *undefined reference to 'yywrap'* (případně podobnou, význam je stejný). Jedná se o to, že vygenerovaný kód analyzátoru vyžaduje definice nejrůznějších funkcí. Funkce *yywrap ()* je volána při dosažení konce vstupního souboru, kdy analyzátor zjišťuje, zda má pokračovat v analýze nějakého dalšího souboru. Obecné

definice této i dalších potřebných funkcí jsou v knihovně libfl.a, která je umístěna ve složce Flexu v podadresáři lib. Pokud tedy programátor nepožaduje vlastní definici, může tuto knihovnu přilinkovat ke zdrojovému souboru analyzátoru a využít zde definované funkce. V tom případě analyzátor po ukončení analýzy souboru předpokládá, že žádný další soubor k analýze nezbývá a ukončí provádění funkce *yylex ()*. A stejného efektu právě dosáhneme vložením řádku *%option noyywrap* do definiční sekce popisu analyzátoru.

6.4 Definiční část

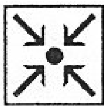
Nyní si ukážeme možnosti, které nám nabízí definiční část vstupního souboru. Je potřeba poukázat na to nutnost pečlivé kontroly odsazení každého řádku, neboť zde platí, že každý řádek, který je odsazen (byť se jedná třeba jen o jednu mezeru), bude beze změny překopírován do výstupního zdrojového souboru. A právě z tohoto důvodu je v příkladu první.l řádek s deklarací proměnných odsazen. Proměnné deklarované v definiční části jsou globální a jsou přístupné v části pravidel i uživatelské části. Stejně tak bude nakládáno s textem sice neodsazeným, ale uzavřeným mezi dvojici %{ a %}, případně /* a */. Tato vlastnost nám umožňuje kromě deklarací proměnných také vkládání hlavičkových souborů či vkládání komentářů. V následujícím příkladu budou všechny řádky překopírovány do výstupního zdrojového souboru.

Řešený příklad 6:



```
        /* Komentář 1 */
%{
/* Komentář 2 */
#include <string.h>
int nejaka_promenna = 0;
%}
%%
```

Jak jsme si již řekli dříve, v části pravidel popisujeme hledané vzory pomocí regulárních výrazů. Abychom si zpřehlednili psaní pravidel, nabízí nám definiční část možnost pojmenovat si regulární výrazy a dále pak používat toto jméno místo vypisování daného regulárního výrazu. Toto pojmenování má tvar *jméno regulární_výraz* a nesmí být odsazeno. Jméno musí začínat písmenem nebo podtržítkem _ a následovat může libovolný počet písmen, čísel či podtržítek _ a pomlček -. Následuje alespoň jedna mezera a samotný regulární výraz.

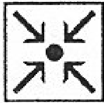


Řešený příklad 7:

CISLICE [0-9]

SLOVO [a-zA-Z]+

V části pravidel pak jsou následující dva řádky ekvivalentní.



Řešený příklad 8:

```
{CISLICE}+        printf ("Nalezl jsem cislo");
[0-9]+            printf ("Nalezl jsem cislo");
```

Poslední možností, kterou nám definiční část nabízí, je přidání položky *%option název_speciální_volby*. Speciální volby lze zapnout také přidáním odpovídajícího parametru při spouštění programu Flex, nicméně některé z dostupných voleb jsou přístupné pouze přes výše uvedený příkaz v definiční části analyzátoru. Kompletní výčet voleb je součástí manuálových stránek programu. Tuto možnost jsme využili v našem prvním příkladu na vynechání volání funkce *yywrap ()* přidáním řádku *%option noyywrap*.

6.5 Část pravidel

V této části určujeme chování analyzátoru při nalezení určitého vzoru. Každé pravidlo se skládá ze vzoru a akce, která se má provést při nalezení daného vzoru ve vstupním textu. Obě části pravidla (vzor akce) se musí

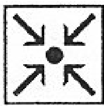
nacházet na jednom řádku a řádek nesmí být odsazen, neboť i zde jsou odsazené řádky zkopírovány do výstupního souboru. Vzor je popsán regulárním výrazem, za ním následuje (oddělena alespoň jednou mezerou) akce ve formě kódu v jazyce C.

Vzory lze popisovat pomocí následujících konstrukcí:

Ahoj	řetězec Ahoj
.	jakýkoliv znak kromě nového řádku
^	začátek řádku
$	konec řádku
[abcd] nebo [a-d]	třída znaků, oběma regulárním výrazům odpovídá každý ze znaků a b c d
[^A-Z]	negace třídy znaků, vyhovuje cokoliv kromě velkých písmen
{jmeno}	pro regulární výrazy pojmenované v definiční části provede expanzi na původní regulární výraz Např. {CISLICE} převede na [0-9] viz. příklad 3
*R**	libovolný počet opakování regulárního výrazu R
R+	alespoň jedno opakování regulárního výrazu R
R(x,y)	regulární výraz opakovaný x-krát až y-krát, jednu z mezí lze vynechat R(x,) R(,y), případně je možný přesný počet opakování R(a)
(R)	uzávorkování lze použít pro upravení priorit
RS	regulární výraz R následovaný regulárním výrazem S
R|S	vyhovuje buď regulární výraz R nebo S
R/S	vyhovuje regulární výraz R právě tehdy, když je následován regulárním výrazem S – tzv. trailing (V tomto případě platí, že S je vráceno zpět na vstup před provedením akce, která pracuje pouze s regulárním výrazem R)
[:digit:]	třída znaků, pro které vrací funkce isdigit () jazyka C hodnotu true (v tomto případě všechny číslice), kompletní seznam tříd je uveden v manuálových stránkách
<<EOF>>	Označuje konec souboru

6.6 Porovnávání vstupního textu

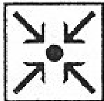
Analyzátor ve vstupním textu hledá řetězce, které odpovídají některému ze vzorů. Zde je důležité poznamenat, že řetězce, které neodpovídají žádnému vzoru, jsou předány na standardní výstup (v našich příkladech tedy vytištěny na obrazovku). Na tuto skutečnost je třeba pamatovat a podle potřeby ji ošetřit. Pokud je nalezen řetězec, který odpovídá více než jednomu vzoru, provede se akce, jejíž regulární výraz odpovídá nejdelšímu řetězci vstupního textu.



Řešený příklad 9:

```
%%
kolo           printf ("Nalezl jsem řetězec  kolo");
kolotoc        printf ("Nalezl jsem řetězec kolotoc");
%%
```

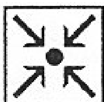
Pokud dáme analyzátoru na vstup soubor, který obsahuje řetězec kolotoc, na obrazovku se vypíše hlášení o nalezení řetězce kolotoc, neboť tento regulární výraz odpovídá delšímu řetězci než regulární výraz kolo.
Pokud je nalezen řetězec, jehož maximální délka odpovídá více vzorům, je použito pravidlo, které je uvedeno v seznamu pravidel dříve.



Řešený příklad 10:

```
%%
[0-9]+         printf ("Nalezl jsem číslo");
1000000        printf ("Nalezl jsem číslo 1000000");
%%
```

Analyzátoru dáme na vstup soubor obsahující pouze číslo 1000000, nicméně se dočkáme pouze oznámení o nalezení nějakého čísla, neboť řetězec 1000000 odpovídá i vzoru [0-9]+, který je uveden v seznamu pravidel dříve, a tedy analyzátor provede akci příslušející tomuto vzoru.
Nyní si ukážeme jeden příklad na tzv. trailing. Jak jsme si již uvedli v přehledu možných konstrukcí regulárních výrazů, jedná se o rozpoznávání regulárního výrazu R, jestliže bezprostředně za ním následuje regulární výraz S (R/S). Analyzátor bude mít na vstupu soubor obsahující desetinná čísla a na výstup nám předá pouze jejich celé části.



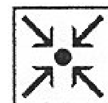
Řešený příklad 11:

```
CISLO [0-9]+
%option noyywrap
%%
{CISLO}/"."{CISLO}      printf ("\nNalezl jsem desetinne cislo, jeho cela cast je
%s",yytext);
"."[0-9]+|\n            /*Ignoruj následek trailingu*/
%%
int main(int argc, char **argv)
{
 ++argv, --argc;  /* skip over program name */
  if ( argc > 0 )
        yyin = fopen( argv[0], "r" );
  else
        yyin = stdin;
  yylex ();
getch();
return 0;
}
```

Zde bychom si měli všimnout především konstrukce *{CISLO}/"."{CISLO},* která nám elegantně umožní dále zpracovat celou část desetinného čísla. V našem případě vypadá trailing tak, že *{CISLO}* zastupuje výraz R a *"."{CISLO}* výraz S. K podrobnostem se dostaneme vzápětí, nicméně nyní stačí říci, že každý rozpoznaný řetězec ukládá analyzátor do proměnné *yytext*. Protože však u trailingu analyzátor pokládá za rozpoznaný řetězec pouze výraz R, uloží se nám do proměnné *yytext* pouze požadovaná celá část desetinného čísla, kterou vzápětí dostaneme na výstupu. Protože však při trailingu analyzátor vrací výraz S zpět na vstup, musíme tuto skutečnost ošetřit. A to má na starosti právě druhé pravidlo *"."[0-9]+|\n,* které zároveň odstraňuje znaky nového řádku (pouze estetický důvod).

6.7 Akce

V případě, že analyzátor rozpozná na vstupu řetězec, který odpovídá určitému vzoru, je následně spuštěna akce, která danému vzoru přísluší. Tou je, jak jsme si již dříve řekli, libovolný kód v jazyce C. Akci můžeme také vynechat, pak budou vstupní řetězce odpovídající danému vzoru jednoduše ignorovány. Použití si ukážeme na odstraňování komentářů ze vstupních souborů.



Řešený příklad 12:

```
%option noyywrap
%%
```

```
\{.*\}              /*ignorujeme komentare Pascalu*/
\/\*.*\*\/  |
\/\/.*\n            /*ignorujeme C/C++ komentare*/
%%
int main(int argc, char **argv)
{
  ++argv, --argc;  /* skip over program name */
   if ( argc > 0 )
           yyin = fopen( argv[0], "r" );
   else
           yyin = stdin;
 yylex ();
getch();
return 0;
}
```

Tento příklad nám ilustroval způsob odstranění komentářů jazyka Pascal a C/C++ ze vstupního souboru. Protože v pravidlech chybí příslušné akce (jsou vloženy pouze komentáře pro lepší přehlednost), jsou řetězce odpovídající daným vzorům ignorovány. Vše ostatní je kopírováno na standardní výstup. Navíc si u druhého pravidla můžeme všimnout symbolu | na místě akce. Toto je symbol pro tzv. rouru (pipe) známý z operačních systémů Unix, který nám poskytuje určité zjednodušení při psaní pravidel. Jeho význam je takový, že akce pro pravidlo, na jejímž řádku se tento symbol vyskytuje, je totožná s akcí následujícího pravidla.

V kódu akcí můžeme navíc využívat speciálních proměnných, maker a funkcí:

char *yytext

Tuto proměnnou jsme s úspěchem použili v příkladu 7 a nyní si o ní povíme něco víc. Jak už tedy víme, do proměnné *yytext* analyzátor ukládá rozpoznaný řetězec. Ačkoliv je proměnná *yytext* implicitně definována jako ukazatel na typ char, je možné ji definovat také jako pole znaků. Toto ovlivňujeme pomocí speciálních příkazů *%pointer* a *%array*, které se vkládají do definiční části. Navíc, při zapnutí režimu kompatibility s programem Lex, bude vždy proměnná *yytext* definována jako pole znaků. Výhodou ukazatele je rychlejší práce analyzátoru (pokud není potřeba velikost *yytext* zvětšovat z důvodu příliš dlouhého vstupního řetězce, v tomto případě dochází k opětovnému porovnávání řetězce od začátku a tedy ke zpomalení práce analyzátoru) a ochrana proti přetečení v případě rozpoznávání dlouhých řetězců. V případě ukazatele můžeme obsah proměnné *yytext* upravovat, nicméně nesmíme zvětšovat její délku. Navíc každé volání speciální funkce *unput ()* způsobí její vymazání. Pokud použijeme typ pole znaků, můžeme s proměnnou *yytext*

pracovat bez omezení, avšak tento typ nelze použít v případě výstupu v jazyce C++. Délka pole znaků je definována konstantou *YYLMAX*, pomocí které také můžeme tuto délku ovlivňovat.

int yyleng uchovává délku aktuálně rozpoznaného řetězce

ECHO kopíruje obsah *yytext* na standardní výstup

BEGIN stav v případě využití stavů nastaví požadovaný stav

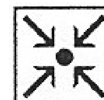
REJECT nařídí analyzátoru spustit akci druhého nejlepšího vzoru

yymore () zajistí, že další rozpoznaný řetězec bude připojen za stávající obsah *yytext* místo toho, aby ho nahradil

yyless (n) vrátí obsah *yytext* na vstup až na prvních n znaků. Proměnná *yyleng* se automaticky upraví na délku n

unput (c)

Vrátí znak c zpět na vstup. Protože se znak vrací vždy na začátek vstupu, je potřeba vracet znaky „odzadu“. V následujícím příkladu nám analyzátor rozpozná ve vstupním souboru řetězce komentar a předá nám na výstup celý zdrojový text upravený o ohraničené komentáře Pascalu.



Řešený příklad 13:

```
%option noyywrap
%%
komentar {
      int i;
      char *yycopy;
      yycopy = strdup(yytext);
      unput('}');
      for(i=yyleng -1; i>=0; i--)
        {
         unput(yycopy[i]);
        }
      unput('{');
      free(yycopy);
            }
\{komentar\}      ECHO;
%%
int main(int argc, char **argv)
{
  ++argv, --argc;  /* skip over program name */
   if ( argc > 0 )
          yyin = fopen( argv[0], "r" );
   else
          yyin = stdin;
yylex ();
getch();
return 0;
```

}

Pomocnou proměnnou *yycopy* jsme použili z důvodu mazání obsahu *yytext* definované jako ukazatel při použití funkce *unput ()*.

input () přečte jeden znak za vstupního souboru

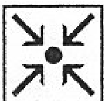
yyterminate () slouží k přerušení analýzy, funkce, která *yylex ()* zavolala, obdrží návratový kód 0, který standardně znamená, že vše bylo dokončeno. Automaticky je *yyterminate ()* voláno na konci vstupního souboru.

6.8 Práce se stavy

Protože lexikální analyzátor je vlastně konečný automat, ukážeme si nyní, jak je v programu Flex realizována práce se stavy. Flex obsahuje mechanizmus pro podmíněné spouštění pravidel. Tedy můžeme mít například pro jeden vzor více pravidel, mezi kterými bude analyzátor vybírat podle toho, ve kterém stavu se právě nachází. Jednotlivé stavy deklarujeme na samostatných řádcích v definiční části. Rozeznáváme dva typy stavů:

- exkluzivní – pokud je analyzátor v některém exkluzivním stavu, vybírá pouze mezi pravidly, které jsou podmíněné daným stavem,
- inkluzivní – v případě, že se analyzátor nachází ve stavu inkluzivním, vybírá mezi všemi nepodmíněnými pravidly a těmi, které jsou podmíněné daným stavem.

Typ stavu určíme v definiční části speciální konstrukcí *%x* (stav exkluzivní) nebo *%s* (stav inkluzivní). Podmíněná pravidla pak vytváříme přidáním jména stavu, uzavřeného mezi znaky < a >, na začátek řádku s pravidlem.



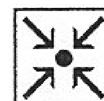
Řešený příklad 14:

```
%x      stav1
%s      stav2
%%
<stav1>  pravidlo1        /* podmíněné  pravidlo
<stav2>  pravidlo2        /* podmíněné  pravidlo
pravidlo3                 /* nepodmíněné  pravidlo
```

V předchozím příkladu má analyzátor ve stavu stav1 k dispozici pouze pravidlo1, protože stav1 je deklarován jako exkluzivní. V případě, že se nachází ve stavu stav2, má na výběr mezi pravidly pravidlo2 a pravidlo3, neboť stav2 je definován jako inkluzivní a tedy zahrnuje i pravidla nepodmíněná žádným stavem.

Při spuštění se analyzátor nachází ve stavu INITIAL, ve kterém může vybírat pouze z nepodmíněných pravidel. Přepínání mezi stavy provádíme pomocí příkazu BEGIN(jmeno_stavu). Příkaz BEGIN můžeme použít mimo akcí také na začátku části pravidel a určit tak explicitně počáteční stav analyzátoru. Tento přechod lze navíc podmínit hodnotou nějaké proměnné a celý příkaz musí být odsazen.

Řešený příklad 15:



```
int nastav_novy_pocatecni_stav;
%x novy_pocatecni_stav
%%
   if (nastav_novy_pocatecni_stav)
   BEGIN(novy_pocatecni_stav);
<novy_pocatecni_stav>  pravidlo
```

Stavy mohou být reprezentovány kromě jména také celým číslem. Například příkazy BEGIN(INITIAL) a BEGIN(0) jsou totožné a oba způsobí přepnutí analyzátoru do implicitního stavu. Pro zjištění čísla aktuálního stavu analyzátoru používáme makro YY_START (případně YYSTATE z důvodu kompatibility s programem Lex).

Pokud chceme sdružit více pravidel pod jeden stav, nemusíme psát před každé z nich název daného stavu. Flex umožňuje následující syntaxi:

```
<stav1>{
        pravidlo1
        pravidlo2
}
```

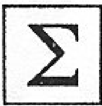
Tento zápis je ekvivalentní:

```
<stav1>  pravidlo1
<stav1>  pravidlo2
```

V případě potřeby nabízí Flex pro práci ze stavy zásobník. Jeho velikost se dynamicky mění a je omezena pouze velikostí dostupné paměti. Používání

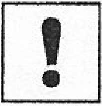
zásobníku stavů musíme explicitně povolit volbou *%option stack* v definiční části. Pro práci se zásobníkem stavů máme k dispozici následující funkce:

- void yy_push_state(int novy_stav) – uloží na vrchol zásobníku stavů aktuální stav a poté přepne do stavu novy_stav,
- void yy_pop_state() – vybere stav z vrcholu zásobníku stavů a přepne do něj,
- int yy_top_state() – vrátí vrchol zásobníku beze změny obsahu zásobníku.



Nejdůležitější probrané pojmy:

- definice pro FLEX
- pravidla pro FLEX
- akce pro FLEX



Úkoly k textu:

1. Naprogramujte lexikální analyzátor pro reálné číslo Pascalovského stylu (tedy obsahuje celou část, může obsahovat desetinnou část a exponent, např. +123.456e-3) pomocí FLEXu.

7 Bison

Cíl:

Po prostudování této kapitoly pochopíte:

- činnost programu BISON
- tvorbu definic a akcí pro BISON

Po prostudování této kapitoly se naučíte:

- používat program BISON pro tvorbu syntaktických analyzátorů

Průvodce studiem

Další nástroj, se kterým se naučíme pracovat, patří do skupiny generátorů syntaktických analyzátorů jazyků. Konkrétně se jedná o Bison ve verzi 1.875 pro operační systém Windows. Aby byl Bison schopen vygenerovat syntaktický analyzátor určitého jazyka, tento musí splňovat určitá kritéria. Především se musí jednat o jazyk, který lze popsat bezkontextovou gramatikou.

Druhy gramatik a rozdíly mezi nimi jsou popsány v úvodní části této práce a tedy se jimi již nebudeme zabývat. Ačkoliv je Bison schopen zpracovat téměř jakoukoliv bezkontextovou gramatiku, je optimalizován pro použití s LALR(1) gramatikami. Tedy musí být jednoznačně definováno, jak postupovat dále pohledem pouze o jeden token dopředu. V praxi tedy popisujeme seskupení jazykových prvků podle syntaxe jazyka a definujeme pravidla pro jejich konstrukci z různých částí. Například v programovacím jazyce C existuje seskupení, které nazýváme „výraz“. Pravidla pro jeho konstrukci pak mohou vypadat takto:

- každé číslo je výraz,
- výraz lze složit ze znaménka „mínus“ následovaného jiným výrazem,

a tak dále. Hojně využíváme princip rekurze, nicméně vždy musí existovat pravidlo, které rekurzi ukončí.

7.1 Instalace

Stejně jako v případě předchozího nástroje Flex je i Bison volně dostupný. Lze jej získat například na domovské stránce programu http://www.gnu.org/software/bison/ ve formě instalačního souboru o velikosti 2,5 MB. Instalační program implicitně kopíruje Bison do složky programu Flex a tedy máme oba nástroje na jednom místě pro snazší práci. Oba programy zabírají dohromady okolo 7MB místa na pevném disku.

V nově vytvořené programové skupině opět nenajdeme spouštěcí soubor, neboť Bison se ovládá stejně jako Flex a tedy jej budeme opět spouštět z příkazové řádky tentokrát pomocí souboru bison.exe. Jako parametr uvádíme jméno vstupního souboru.

Bison – schéma práce

Stejně jako v případě programu Flex se jedná o čistě textový soubor tentokrát s obecně dodržovanou příponou .y. Standardní výstupní soubor, který obsahuje hlavní funkci *yyparse ()*, nese stejný název jako vstupní soubor, pouze přípona .y je nahrazena .tab.c. Výstupní soubor obsahuje zdrojový kód vytvářeného syntaktického analyzátoru v jazyce C. Na možnosti generování výstupu v jazyce C++ se podle autorů programu pracuje, nicméně současná verze toto zatím neumožňuje.

7.2 Volitelné parametry

Při spouštění programu Bison máme k dispozici celou řadu volitelných parametrů, z nichž ty nejdůležitější jsou uvedeny v následujícím seznamu:

-d	-vytváří speciální soubor s definicemi maker popisujících jména typů tokenů užitých v gramatice. V případě, že je funkce lexikálního analyzátoru yylex (), definována v samostatném souboru, vkládáme tento soubor jako hlavičkový.
-h	-vypíše seznam všech dostupných voleb.
-k	-ve výstupním souboru vytvoří pole *yytname* obsahující jména tokenů. Položka *yytname[i]* obsahuje vždy jméno tokenu, jehož vnitřní číselná reprezentace je *i*.
-o	-specifikuje jméno výstupního souboru.
-p*předpona*	-upravuje jména výstupních souborů. Jména jsou nastavena jako kdyby byl definiční soubor pojmenován *předpona*.y.
-v	-vytvoří speciální soubor s příponou .output, ve kterém je popis překladu včetně všech upozornění.
-V	-vypíše verzi programu Bison.
-y	-nastaví jména výstupních souborů tak, aby odpovídala konvenci programu Yacc (y.tab.c, y.output, y.tab.h).

7.3 Formát vstupního souboru

%{

Definice – jazyk C

%}

Definice – Bison

%%

Pravidla

%%

Uživatelský kód v jazyce C

Jak vidíme, struktury vstupních souborů pro Flex i Bison jsou si velmi podobné, což jistě usnadňuje programátorům práci. V první části definic (uzavřené mezi %{ a %}) máme možnost definovat makra a deklarovat funkce a proměnné, které poté využijeme v pravidlech. Dále lze na tomto místě připojovat hlavičkové soubory. Druhou část definic využíváme pro deklarace

terminálních a neterminálních symbolů, definování priorit a podobně. Část pravidel obsahuje vždy alespoň jedno pravidlo gramatiky. Poslední část se řídí stejnými pravidly jako v případě programu Flex, tedy je celá zkopírována do výstupního souboru a slouží k definování uživatelských funkcí a podobně.

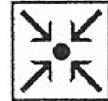
7.4 Symboly

Symboly máme při práci s programem Bison na mysli syntaktické struktury jazyka. Jak již víme, rozdělujeme je na dva typy.
Terminální symboly reprezentují třídy syntakticky ekvivalentních tokenů. Standardně je zapisujeme velkými písmeny (INTEGER, IF). Bison využívá pro jejich reprezentaci číselný kód, což je také důvod, proč funkce *yylex ()* lexikálního analyzátoru vrací hodnotu integer. Tato číselná hodnota reprezentuje právě číselný kód naposledy přečteného tokenu.
Neterminální symboly zastupují skupiny symbolů, které jsou vytvářeny podle pravidel gramatiky, a zapisujeme je malými písmeny (vyraz, deklarace). Jména symbolů mohou obsahovat písmena, číslice (nesmí však číslicí začínat) a znak podtržítka.

7.5 Sémantika jazyka

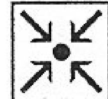
Gramatická pravidla popisují pouze syntax daného jazyka. Každý terminální i neterminální symbol si nese svou sémantickou hodnotu s sebou. Bison implicitně předpokládá, že sémantické hodnoty všech symbolů jsou celočíselné (typ *integer*). Toto však postačuje pouze pro jednodušší příklady, a tak máme samozřejmě možnost vlastního nastavení. Typ sémantické hodnoty symbolů je reprezentován makrem YYSTYPE. Pokud chceme provést změnu na jiný typ, poslouží nám k tomu příkaz *#define YYSTYPE nova_hodnota*.
Nicméně v praxi je samozřejmě nedostatečné, aby měly všechny symboly pouze jedinou možnou sémantickou hodnotu. Proto se nabízí možnost definovat YYSTYPE jako soubor více typů, ze kterých pak vybíráme podle potřeby.

Řešený příklad 16:



```
%{
#define YYSTYPE double        /* všechny symboly budou typu double místo
implicitního int */
%}
%%
```

Řešený příklad 17:



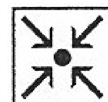
```
%union {        /*definování YYSTYPE jako union */
int  cele_cis;
double  real_cis;
char  *retez;
}
%%
```

Definice makra YYSTYPE se provádí v definiční části vstupního souboru pro Bison. V případě, že definujeme YYSTYPE jako union, je nutné definovat všechny možné sémantické datové typy symbolů a dále pak pro každý symbol gramatiky vybrat jeden z těchto typů.

7.6 Definiční část

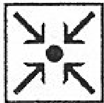
Zaměříme se pouze na část definic pro Bison, neboť první část definic má stejný význam jako v programu Flex. Ukážeme si, jak definovat symboly gramatiky a datové typy sémantických hodnot. První důležitou věcí je nutnost definovat všechny terminální symboly kromě jednoznakových (‘+’, ‘*’, ‘a’, ‘b’ apod.), aby jim mohla být přiřazena čísla pro vnitřní reprezentaci. Jednoznakové tokeny již čísla mají, proto není nutné je zde definovat. Neterminální symboly je potřeba definovat pouze v případě, že chceme explicitně určit datový typ jejich sémantické hodnoty.
Terminální symboly deklarujeme pomocí příkazu *%token jméno_typu_tokenu*, případně můžeme také přímo určit vnitřní číselnou reprezentaci. Pokud k tomu však nemáme zvláštní důvod, je toto vhodnější nechat na Bisonu.

Řešený příklad 18:



```
%token CISLO
%token IF 100
%%
```

V případě, že jsme definovali více možných sémantických typů symbolů, deklarujeme terminální symboly následovně:



Řešený příklad 19:

```
%union {
int  cele_cis;
double  real_cis;
char  *retezec;
}
%token <cele_cis> CCISLO
%token <retetec> RETEZEC
%%
```

V případě, že je nutné deklarovat také neterminální symboly, využíváme následující zápis:

%type <typ> neterminál1 neterminál2…,

kde *typ* vybereme z YYSTYPE a více neterminálních symbolů se stejným sémantickým typem lze zapsat za sebou oddělené mezerami. Jelikož gramatika má vždy nějaký startovací neterminální symbol, můžeme jej explicitně určit:

%start symbol,

pokud tak neučiníme, Bison pokládá za startovací neterminální symbol první neterminální symbol uvedený v sekci pravidel.

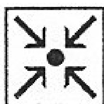
V případě tokenů operátorů lze nastavit jejich asociativitu a vzájemnou prioritu. V tom případě máme pro deklaraci k dispozici místo *%token* následující příkazy:

%left symboly,

%right symboly,

%nonassoc symboly,

se kterými pracujeme stejně jako s *%token*, včetně možnosti definovat explicitní typ sémantické hodnoty. Pořadí deklarací jednotlivých tokenů zároveň určuje jejich vzájemnou prioritu. Tokeny uvedené na jednom řádku mají stejnou prioritu a čím výše je token uveden, tím nižší prioritu má.



Řešený příklad 20:

```
 %left ‘<’ ‘>’ ‘=’ “<=” “>=”
%left ‘+’ ‘-‘
%left ‘*’ ‘/’
%right ‘^’
%%
```

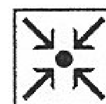
Takto nadefinované tokeny operátorů dodržují obecně známá pravidla pro asociativitu a vzájemnou prioritu.

7.7 Část pravidel

Tato část obsahuje samotný popis gramatiky. Jak jsme si již dříve řekli, musí se zde nacházet alespoň jedno pravidlo. Každé pravidlo gramatiky začíná neterminálním symbolem, následuje dvojtečka a za ní posloupnost terminálních a neterminálních symbolů, ze kterých první neterminální symbol skládáme. Dále následuje volitelná akce, která se provede, když dojde k použití daného pravidla a vše je ukončeno středníkem. Obecný tvar pravidel gramatiky tedy vypadá následovně:

neterminál: části_neterminálu { akce };

Řešený příklad 21:



%token CISLO

%%

vyraz: CISLO {kód akce};

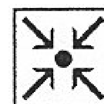
vyraz: CISLO '+' CISLO {kód akce};

vyraz: CISLO '-' CISLO {kód akce};

%%

V předchozím příkladu jsme nadefinovali jeden terminální symbol CISLO, který je zároveň nejjednodušším výrazem a pomocí pravidel gramatiky jsme určili způsob, jak z něj mohou vznikat výrazy pomocí sčítání a odečítání. V případě, že máme několik pravidel pro jeden neterminální symbol, můžeme zápis zpřehlednit:

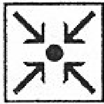
Řešený příklad 22:



```
%token CISLO
%%
vyraz:   CISLO             {kód akce}
| CISLO '+' CISLO          {kód akce}
| CISLO '-' CISLO          {kód akce};
%%
```

V tomto případě vložíme koncový středník až za poslední pravidlo. V rámci pravidel můžeme libovolně vkládat mezery pro zpřehlednění, protože

tyto jsou využívány pouze pro oddělení jednotlivých symbolů. V případě potřeby je možné vytvořit pravidlo, které neobsahuje posloupnost částí, ze kterých přepisovaný neterminální symbol skládáme. V tom případě přepisovaný neterminální symbol odpovídá prázdnému řetězci. V následujícím příkladu si nadefinujeme seznam příjmení, který může být buď prázdný nebo se skládá z jednotlivých příjmení oddělených čárkou.



Řešený příklad 23:

```
%token PRIJMENI
%%
seznam_jmen: /*vynecháváme*/
                  | jmena;
jmena: PRIJMENI
         | jmena','PRIJMENI;
%%
```

Při sestavování pravidel se nevyhneme použití rekurze, neboť je to jediný způsob, jak definovat posloupnost opakujících se částí. Rekurzivní pravidlo je takové, v němž se levá strana (přepisovaný neterminální symbol) vyskytuje také na straně pravé. Rozlišujeme následující druhy rekurzí:

- levá rekurze

seznam_hostu: HOST

| seznam_hostu',' HOST;

- pravá rekurze

seznam_hostu: HOST

| HOST',' seznam_hostu;

- nepřímá rekurze

[doplnit]

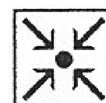
Obecně platí, že cokoliv lze definovat pomocí pravé rekurze, lze definovat také pomocí levé a naopak. Nicméně je vždy velmi vhodné použít rekurzi levou, která má daleko menší paměťové nároky.

7.8 Akce

V programu Bison zapisujeme akce v pravidlech vždy uzavřené do složených závorek, díky čemuž mohou být rozepsána na více řádcích. Akce

daného pravidla se spustí ve chvíli, kdy dojde k aplikaci daného pravidla. Akci můžeme vkládat na jakékoliv místo pravidla, kde bude také provedena. Většina pravidel má pouze jednu akci, která bývá uvedena na konci pravidla. Nejčastějším typem akce, která se provádí, je výpočet sémantické hodnoty neterminálního symbolu pomocí sémantických hodnot jeho částí. Na sémantické hodnoty se odkazujeme pomocí proměnné *$n*, kde n označuje n-tý symbol na pravé straně pravidla. Speciální proměnná *$$* uchovává sémantickou hodnotu neterminálního symbolu na levé straně pravidla.

Řešený příklad 24:



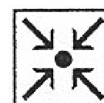
```
%token CISLO
%%
vyraz: CISLO {$$ = $1;}
        | vyraz '+' vyraz {$$ = $1 + $3;} /* $2 přísluší symbolu '+'*/
        | vyraz '-' vyraz {$$ = $1 - $3;};
%%
```

V případě prvního pravidla odpovídá sémantická hodnota výrazu hodnotě samotného čísla, které jej tvoří. V dalších případech sčítáme (odečítáme) hodnoty jednotlivých (pod)výrazů.

Pokud používáme oddělovače | pro zápis více pravidel pro jeden neterminální symbol, je třeba si uvědomit, že akce se vztahuje vždy jen k tomu pravidlu, v jehož těle je uvedena. V tom je zde rozdíl oproti zápisu akcí v programu Flex, kde se oddělovač | používá v případech, že chceme mít jednu akci společnou pro více pravidel. Pokud v pravidle neuvedeme žádnou akci, Bison doplní implicitní akci *$$ = $1*, tedy sémantická hodnota prvního symbolu na pravé straně se stane sémantickou hodnotou celého pravidla (neterminálního symbolu na levé straně). Toto platí pouze pokud odpovídají datové typy daných symbolů. Pro pravidla s prázdnou pravou stranou neexistuje implicitní akce a tedy je ji třeba vždy určit explicitně, pokud je sémantická hodnota daného pravidla požadována.

Proměnné *$$* a *$n* mají vždy stejný datový typ jako symbol, na který jsou v daném pravidle navázány. Nicméně existuje možnost jejich přetypování následující formou:

Řešený příklad 25:



```
%union {
                int cele_cislo;
                double real_cislo;
}
```

```
%token <cele_cislo> CISLO
%type <cele_cislo> vyraz
%%
vyraz: CISLO
…
```

V tomto případě můžeme například na hodnotu tokenu CISLO odkazovat také jako na realné číslo pomocí konstrukce *$<real_cislo>1*.

Ačkoliv se nejčastěji setkáváme s akcemi na konci pravidel, v některých případech může být výhodné umístit akci uvnitř pravidla. Pro tyto akce platí stejná forma zápisu jako pro akce na konci pravidel s jedním omezením. Akce uvedené uvnitř pravidla se provádějí dříve než analyzátor rozpozná ostatní komponenty uvedené za touto akcí, a tedy není možné se na tyto komponenty v dané akci odkazovat. Ke komponentám uvedeným před akcí přistupujeme standardně pomocí proměnných *$n*. Samotná akce se chová jako komponenta pravidla, což je nutné mít na paměti, pokud na konec pravidla umístíme další akci, ve které budeme využívat sémantických hodnot komponent. Akce zároveň může mít vlastní sémantickou hodnotu, kterou si nastaví pomocí proměnné *$$*. Z toho také vyplývá, že v tomto typu akce nelze nastavit sémantickou hodnotu celého pravidla, neboť proměnná *$$* v tomto případě slouží k jinému účelu. Vzhledem k tomu, že akce uvnitř pravidla nemají vlastní jméno, není možné deklarovat typ jejich sémantické hodnoty dopředu. Proto jej musíme ručně specifikovat při každém odkazování na hodnotu dané akce pomocí konstrukce na přetypování.

7.9 Pozice symbolů

V určitých případech může být vhodné znát přesnou polohu zpracovávaných symbolů ve vstupním souboru. Například ve chvíli, kdy analyzátor objeví syntaktickou chybu je žádoucí, aby byl uživatel informován nejen o faktu, že taková chyba existuje, ale také o místě výskytu dané chyby ve zdrojovém souboru. Ze všeho nejdříve je třeba určit, v jakém tvaru budeme s pozicí symbolu pracovat. Tedy definujeme datový typ této hodnoty, což je jednodušší v porovnání se sémantickými hodnotami, protože všechny symboly mají vždy stejný formát své pozice. Pozici daného symbolu zpracováváme

pomocí makra YYLTYPE. Pokud si toto makro nedefinujeme sami, Bison využívá předdefinovaný tvar v následujícím tvaru:

typedef struct YYLTYPE

{

int first_line;

int first_column;

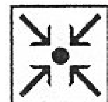
int last_line;

int last_column;

} YYLTYPE;

Pro práci s pozicí symbolu máme k dispozici několik konstrukcí, které jsou podobné konstrukcím pro práci se sémantickými hodnotami symbolů. Pozice neterminálního symbolu na levé straně pravidla je přístupná přes konstrukci *@$* a k pozicím komponent na pravé straně přistupujeme pomocí @*n*, kde n je opět pořadí daného symbolu na pravé straně pravidla.

Řešený příklad 26:



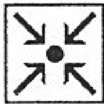
```
vyraz:   vyraz '/' vyraz
           {
             @$.first_column = @1.first_column;
             @$.first_line = @1.first_line;
             @$.last_column = @3.last_column;
             @$.last_line = @3.last_line;
             if ($3) $$ = $1 / $3;
             else
              {
                $$ = 1;
                fprintf (stderr,
                        "Deleni nulou, radek %d,pozice %d-radek %d,pozice %d",
                        @3.first_line, @3.first_column,
                        @3.last_line, @3.last_column);
              }
           }
```

V tomto příkladu provádíme kontrolu na dělení nulou v podílu dvou výrazů. Na počátku nastavíme hodnotu pozice neterminálního symbolu *vyraz* (z levé strany pravidla) podle pozic krajních symbolů, které na něj přepisujeme. Tedy začátek *@$* nastavíme na začátek @*1* a konec nastavíme na konec @*3*. Posléze testujeme sémantickou hodnotu *$3* a pokud je nenulová, spočítáme podíl *$1* a *$3*. V případě dělení nulou toto oznámíme na standardní chybový výstup společně s udáním pozice chyby ve vstupním souboru.

Stejně jako pro sémantické hodnoty symbolů i v případě hodnot pozic existuje implicitní akce, která se provede při každé aplikaci pravidla. Je shodná s akcí v předchozím příkladu, tedy nastaví počátek *@$* na počátek prvního symbolu

z pravé strany pravidla a konec *@$* na konec posledního symbolu z pravé strany pravidla. Předchozí příklad tedy můžeme zjednodušit takto:

Řešený příklad 27:



```
vyraz:    vyraz '/' vyraz
{
 if ($3) $$ = $1 / $3;
 else
{
$$ = 1;
fprintf (stderr,
"Deleni nulou, radek %d,pozice %d-radek %d,pozice %d",
 @3.first_line, @3.first_column,
 @3.last_line, @3.last_column);
}
}
```

Pokud nám z nějakého důvodu implicitní akce nevyhovuje, můžeme si ji upravit podle potřeb. Požadovanou změnu provedeme úpravou makra YYLLOC_DEFAULT, které má následující výchozí tvar:

```
# define YYLLOC_DEFAULT(Current, Rhs, N)
        do
         if (N)
        {
        (Current).first_line   = YYRHSLOC(Rhs, 1).first_line;
        (Current).first_column = YYRHSLOC(Rhs, 1).first_column;
        (Current).last_line    = YYRHSLOC(Rhs, N).last_line;
        (Current).last_column  = YYRHSLOC(Rhs, N).last_column;
        }
        else
        {
        (Current).first_line = (Current).last_line =
        YYRHSLOC(Rhs, 0).last_line;
        (Current).first_column = (Current).last_column =
        YYRHSLOC(Rhs,0).last_column;
        }
         while (0)
```

První parametr *Current* určuje pozici celého pravidla (výsledného neterminálního symbolu na levé straně pravidla), *Rhs* určuje pozice všech symbolů na pravé straně pravidla a *N* je velikost pravé strany pravidla.

YYRHSLOC (Rhs, k) určuje pozici k-tého symbolu v *Rhs*.

7.10 Pracovní algoritmus

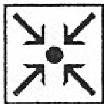
Formální reprezentací syntaktického analyzátoru je zásobníkový automat. Lexikální analyzátor posílá syntaktickému analyzátoru tokeny a ten je ukládá na zásobník spolu s jejich sémantickými hodnotami. Operace přesunu tokenu na zásobník se nazývá **přesun**. Druhou operací se zásobníkem je **redukce**. K té dochází v případě, že několik posledních tokenů či neterminálních symbolů v zásobníku odpovídá pravé straně některého z pravidel gramatiky. Tyto jsou následně na zásobníku nahrazeny levou stranou odpovídajícího pravidla. Spolu s redukcí je zároveň provedena akce příslušného pravidla. Pomocí operací přesun a redukce se analyzátor snaží zredukovat celý vstup na jediný neterminální symbol, který označujeme jako startovací symbol gramatiky. Tento způsob práce pak nazýváme syntaktickou analýzou zdola nahoru. K redukci symbolů na zásobníku nedochází vždy ihned po tom, co je nalezeno pravidlo, které by dané symboly redukovalo. Místo toho se uplatňuje strategie založená na pohledu o jeden token vpřed a teprve na základě tohoto tokenu dochází k rozhodnutí analyzátoru zda redukovat či ne.

V okamžiku přečtení tokenu nedochází k jeho okamžitému přesunu na zásobník. Token se nejdříve stane tzv. sledovaným tokenem (*look-ahead token*) a zatím se na zásobník neumístí. Možné redukce symbolů na zásobníku se provádějí v závislosti na typu tohoto sledovaného tokenu. Může tedy dojít k situaci, kdy analyzátor pozdrží aplikaci určitého pravidla, protože podle sledovaného tokenu pozná, že v budoucnu by došlo například k syntaktické chybě.

Jedním druhem možných problému je tzv. **konflikt přesun/redukce**. Jedná se o situaci, kdy je možné provést jak redukci symbolů na zásobníku, tak přesun sledovaného tokenu na zásobník a analyzátor nemá k dispozici žádné další informace, podle kterých by se rozhodl. K této situaci dojde například při existenci následujícího pravidla v gramatice:

podmínka_if: *IF vyraz THEN prikaz*
| IF vyraz THEN prikaz ELSE prikaz;

V okamžiku, kdy syntaktický analyzátor dostane token ELSE a ten se stane sledovaným tokenem, je na zásobníku posloupnost symbolů, které lze redukovat pomocí prvního pravidla. Nicméně analyzátor má navíc informaci, že může rovněž provést přesun tokenu ELSE na zásobník, protože posléze bude možné aplikovat druhé pravidlo. Bison je navržen tak, aby jím vygenerované syntaktické analyzátory upřednostnily přesun před redukcí.
Nicméně existuje případ, kdy lze toto upřednostnění přesunu před redukcí změnit. Jedná se o případy matematických výrazů, kde je důležitá priorita operátorů.

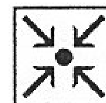


Řešený příklad 28:

```
vyraz: vyraz ‘+’ vyraz
        | vyraz ‘*’ vyraz
        | vyraz ‘!’
        | CISLO;
```

Nyní si vezměme například vstup 4*7+2. Syntaktický analyzátor dostal na vstup tokeny 4, *, 7 a přesunul je na zásobník. Nabízí se redukce pomocí prvního pravidla. Nyní obdrží analyzátor token ‘+’ a náhle se dostane do situace, kdy jak operace redukce tak operace přesun povedou k úspěšné analýze. Rozdíl bude pouze ve výsledku. Pokud přesuneme ‘+’, tento bude v budoucnu redukován dříve a tedy výsledek bude interpretován jako 4*(7+2). V případě, že provedeme nejdříve redukci 4*7, bude výsledek odpovídat (4*7)+2. V tomto konkrétním případě je tedy žádoucí upřednostnit redukci, protože priorita operátoru * je vyšší než priorita operátoru +.
Pokud by však byl vstupem například výraz 4+7!, po přečtení tokenu ‘!’ je třeba provést jeho okamžitý přesun na zásobník, aby byla i v tomto případě dodržena priorita operátorů.
Dalším typem problémů je tzv. **konflikt redukce/redukce**. Jedná se o problém, který vyvstane v okamžiku, kdy lze na nějaký vstup aplikovat více pravidel. Bison je navržen tak, že aplikuje pravidlo, které je v gramatice uvedeno dříve, nicméně toto řešení není ideální a je proto vždy lepší takový konflikt vyřešit úpravou gramatiky.

Řešený příklad 29:

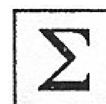


```
veta:   /*žádná věta*/
        | veta slova;
        | veta cisla;
slova:  /*žádné slovo*/
        | slova slovo;
cisla:  /*žádné číslo*/
        | cisla cislo;
```

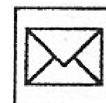
Zde je pokus o definici věty, která může obsahovat slova a čísla. Definice jednotlivých neterminálních symbolů jsou v pořádku, nicméně jejich spojením vzniká problematická nejednoznačnost. Jednou z možností, jak vyřešit tento problém, je následující přepis:

veta: */*žádná věta*/*
| veta slova;
| veta cisla;
slova: slovo
| slova slovo;
cisla: cislo
| cisla cislo;

Nejdůležitější probrané pojmy:



- definice pro BISON
- pravidla pro BISON
- akce pro BISON



Korespondenční úkol:

Sestrojte bezkontextovou gramatiku a Backusovu-Naurovou formu pro jednoduchý programovací jazyk složený ze seznamu řádků s příkazy. Řádek je uvozen návěštím ve tvaru X: příkaz, kde X je přirozené číslo. Lze používat identifikátory, které začínají písmenem anglické abecedy a obsahují libovolně mnoho písmen a číslic. Příkazy které se mohou použít jsou následující:

- přiřazovací příkaz tvaru X = aritmetický výraz, kde X je identifikátor a aritmetický výraz je výraz obsahující operandy – identifikátory a dále reálná čísla, operace sčítání (+), odčítání (-), násobení (*) a dělení (/) a také umožňují vnořovat podvýrazy pomocí závorek.
- Nepodmíněný skok tvaru > X, kde X je návěští řádku, na který se má skočit.
- Podmíněný skok tvaru ?X$Y, kde X je identifikátor a Y je návěští a význam tohoto příkazu je, že se skočí na Y pouze pokud hodnota identifikátoru X je nula.
- Příkaz ukončení běhu programu - !

Pro tento jazyk naprogramujte (pomocí některé z metod z tohoto textu) interpretační překladač tohoto jazyka. Vytvořte několik příkladů, na kterých se funkčnost vašeho interpretu ukáže.

```
10:X=5
20:Y=1
30:Y=X*Y
40:X=X–1
50 ?X$70
60:>30
70:!
```

8 Případová studie – reprezentace a překlad logických výrazů

Po prostudování této kapitoly pochopíte:

- **co je syntaktický strom**
- **kde se používá v reálných aplikacích,**
- **jaké základní typy prohledávání stromu známe,**

Klíčová slova této kapitoly:

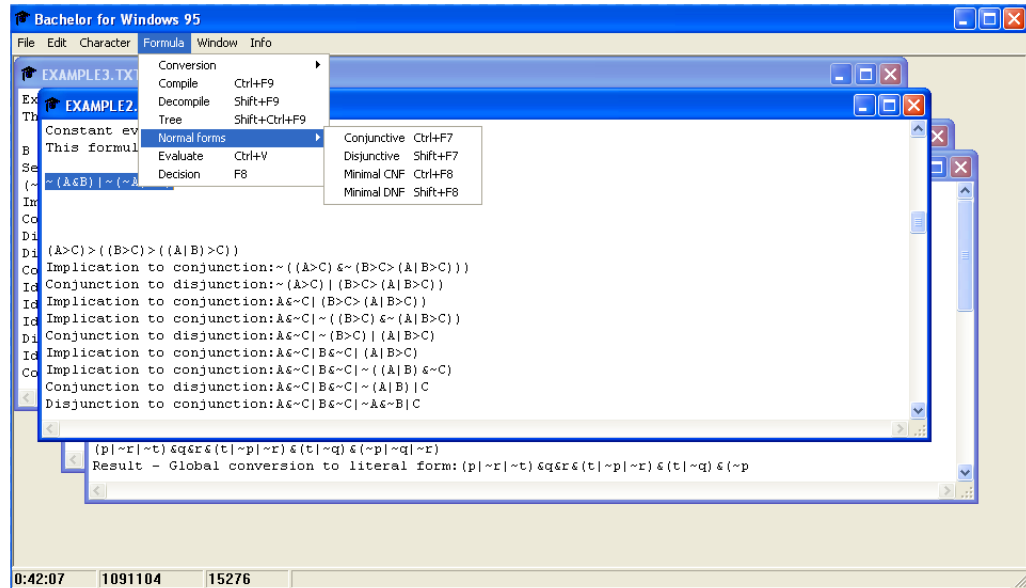
Bachelor, výrokové formule, syntaktický strom, formální překlad.

Průvodce studiem

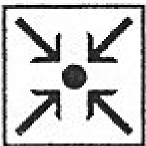
Jak jsme již pochopili v předchozí kapitole, není zápis v klasickém infixním formátu pro počítač příliš výhodný, i když my lidé jsme na něj zvyklí a umíme s ním díky výuce matematiky dobře pracovat. To platí o výrazech i strukturovaných kódech všeho druhu - programy, texty, aritmetické a logické výrazy. Ukázali jsme si způsob, jak lze jednoduše vytvořit jinou notaci (v podstatě jde jen o pořadí operátorů a operandů), kterou už počítač dokáže relativně lehce zpracovat (s pomocí zásobníku). Existuje však mnohem univerzálnější a pro algoritmizaci složitých procesů velmi vhodná metoda a to je reprezentace pomocí syntaktických stromů.

Syntaktický strom je v podstatě orientovaným grafem s ohodnocenými uzly neobsahujícím cykly a u kterého, lze určit kořen stromu (jakýsi nejvyšší prvek). Každý uzel, pak může mít své potomky k nimž vedou hrany a ty uzly, které potomky nemají se nazývají listy stromu (graf tedy opravdu připomíná

živý strom). Pro jednoduchost budeme stromy reprezentovat textově - jde o výstupy z již zmiňovaného doplňku vytvořeného pro čtenáře v rámci aplikace Bachelor (Habiballa, 09a). Ovšem pozor, tento doplněk je obsažen v nabídce Formula, Tree jen ve verzi pro Windows 32-bit, tedy WinBch95.exe.

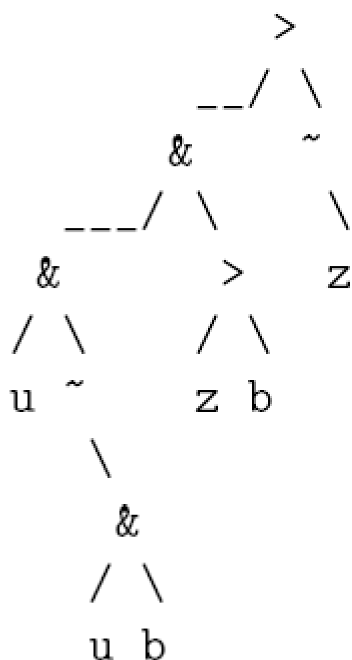


Obrázek 1: Program Bachelor



Příklad 1.

Mějme logickou formuli - u&~(u&b)&(z>b)>~z.



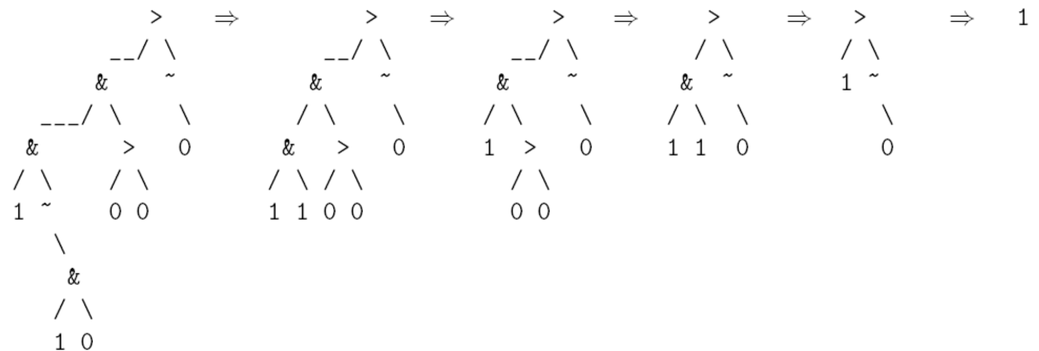
Obrázek 2: Strom formule u&~(u&b)&(z>b)>~z.

Tento strom výstižně a hlavně pro počítač dostatečně jednoduše postihuje celou hierarchii formule. V kořeni stromu je spojka implikace, což by počítač z infixní formy určoval velmi složitě (závorky, priority operátorů), zatímco pracuje-li se stromem, vše je z hierarchické struktury stromu jasné. Implikace má jako podstromy (spojené znaky / a \ a pomocnými znaky _) konjunkci složené formule a negace atomu z. Strom vůbec nemusí obsahovat závorky, neboť priorita operátorů je obsažena už v samotném stromu a není důvod ji narušovat nějakým speciálním symbolem jako jsou závorky. Algoritmy pro manipulace, změny a výpisy stromu mohou být navíc velmi chytře popsány rekurzivními algoritmy, které jsou pak velmi jednoduché (krátké).

Například pokud bychom znali hodnoty logických proměnných u, z a b, pak spočtení hodnoty výrazu se dá popsat jednoduchým rekurzivním pravidlem P:
Pokud je uzel listem vrať hodnotu proměnné,

Pokud uzel není listem, spočítej hodnoty všech potomků uzlu pravidlem P a pak tyto hodnoty zpracuj operací, která je obsažena v uzlu, a vrať výsledek (negaci logické konstanty spočítej přímo).

Podívejme se na to ilustrativně na samotném stromu. Vezměme stejný strom, pouze dosaďme hodnoty proměnných do listů u=1,z=0,b=0.



Tento příklad ilustruje operace se stromem, který nakonec vede k logické konstantě true (tudíž dedukce je správná).

8.1 Konstrukce stromu

Viděli jsme, že práce se stromem je pro počítač velmi příjemná a dá se formulovat takřka triviálními algoritmy (a bude platit i u mnohem složitějších operacích, jak uvidíme dále). Otázkou zůstává, jak s pomocí již známého rekurzivního sestupu a postfixového způsobu zápisu výrazu sestrojit syntaktický strom. Principiálně to není příliš složité, pouze technické zpracování vyžaduje dovednosti při programování dynamických datových struktur a práci s ukazateli. Využijeme stejného algoritmu se zásobníkem, jakým bychom vyhodnocovali aritmetický výraz, pouze změníme vykonávané akce. Místo vložení číslice do zásobníku vytvoříme dynamickou proměnnou typu záznam, kterou vložíme do zásobníku. Tento záznam reprezentuje list stromu. Pokud v nějakém místě programu bychom podle starého postupu

vkládali operátor, pak namísto toho vytvoříme opět dynamický záznam. Ten obsahuje odkaz na své nejvýše dva potomky (podvýrazy spojené logickou spojkou) - těmito odkazy ukážeme na poslední dva prvky v zásobníku, které předtím vybereme. Místo nich pak vložíme hotový záznam se symbolem daného operátoru. Na konci budeme mít ze správně utvořeného logického výrazu mít provázaný syntaktický strom a jeho kořen bude k dispozici na vrcholu zásobníku. Následující fragment kódu aplikace Bachelor ukazuje, jak vypadá datová struktura pro uložení jednoho uzlu stromu TSTreeNode a dále jak vypadá zapouzdřovací záznam pro dočasné uložení do zásobníku (jakýsi öbal", který potřebujeme pro uchování v zásobníkové paměti).

```
type

     PSTreeNode=^TSTreeNode;   {uzel stromu}
     TSTreeNode=record
      character:char;          {symbol : logická spojka,
                                proměnná      nebo      logická
konstanta}
      prior:byte;               {priorita symbolu - potřebujeme
ji pouze
                ke  zpětnému  výpisu do  infixní  podoby  -  tvorba
závorek}
      neg:boolean;             {příznak zda je uzel negován
                - nevytváříme zbytečně v programu další větev}
      left:PSTreeNode;              {ukazatel  na  levý  podstrom
(podvýraz) - potomka}
      right:PSTreeNode;            {ukazatel  na  pravý  podstrom
(podvýraz) - potomka}
     end;


     PSStackNode=^TSStackNode; {obal pro uchování v zásobníku}
     TSStackNode=record        {během překladu do stromu}
      node:PSTreeNode;         {ukazatel na vložený uzel}
      next:PSStackNode;         {ukazatel na následující obal v
zásobníku}
     end;
```

Princip tvorby stromu během rekurzivního sestupu si ukážeme na fragmentech analyzátoru. Popsat celý analyzátor by zabralo mnoho stran a principiálně nejde o odlišný případ, jaký jsme již předvedli na aritmetickém výrazu. Čtenář má navíc možnost se podívat do zdrojových kódů aplikace Bachelor. Ke všem neterminálům jsou v souladu s definicí BNF našeho jazyka sestrojeny procedury rekurzivního sestupu.

```
...
procedure Exp3;
 begin
  if Error=OK then
    begin
      Exp4;
      while (ch='|')and(error=OK) do
        begin
         Getchar;
         Exp4;

         if error=OK then Action('|',3);

        end;
    end;
 end;
...
procedure ICE;
 begin
 if error=OK then
  begin
   case ch of
    '~':begin
          Getchar;
          ICE;
          VPSStack^.node^.neg:=not(VPSStack^.node^.neg);
        end;
    'a'..'z','A'..'Z','0'..'1':
        begin
          Put(ch,6);
```

```
          Getchar;
        end;
    '(':begin
          Getchar;
          Exp1;
          if (ch<>')')and(error=OK) then Error:=missbra;
          Getchar;
        end
    else Error:=missexp;
    end;
  end;
 end;
```

Vidíme, že vložení listu do zásobníku nám realizuje procedura Put, která vytvoří daný uzel a jeho obal s příslušným znakem a prioritou. Pokud nám přijde znak negace, změníme u posledního vloženého atomu negaci na opačnou (teoreticky může obsahovat libovolné, množství negací za sebou). V programu je definováno mnoho chyb, které může výraz obsahovat - zde například chyba missexp vyjadřuje chybějící podvýraz a missexp chybějící uzavírací párovou závorku. Navázání vytvoření a navázání potomků uzlu, který není list (operátor), nám zajistí procedura Action.

```
procedure  Put(var  chp:char;pr:byte);    {vytvoření  a  vložení
atomu}
    begin
      new(pom); {alokace nového dynamického záznamu - obalu}
      pom^.next:=VPSStack;  {původní  vrchol  zásobníku  musíme
navázat na nový}
      VPSStack:=pom; {nový obal je první v zásobníku - vrchol}
      new(pom^.node); {v obalu vytvoříme uzel stromu}
      VPSStack^.node^.character:=chp; {uzel má požadovaný znak}
      pom^.node^.left:=nil;
      pom^.node^.right:=nil;  {levý  i  pravý  podstrom  atom  -
proměnná
                    nebo konstanta - nemá! tedy ukazuje na nil}
      pom^.node^.prior:=pr;    {nastavíme    prioritu    podle
požadavku}
      pom^.node^.neg:=false; {prvotní vytvoření nedělá negaci}
```

```
    end;
...
procedure   Action(chr:char;pr:byte);        {vlož  operátor  do
zásobníku}
 begin
   Put(chr,pr); {vlož do zásobníku nový obal s uzlem
                o požadovaném znaku operace a prioritě}
   pom:=Cut;  {vyber  operátor  ze  zásobníku  -  budeme  s  ním
pracovat}
   pom^.node^.right:=VPSStack^.node;  {navaž  první  uzel  na
zásobníku
                    jako pravý podstrom - pozor v zásobníku je
pořadí obrácené}
   garbage:=Cut;  {vyber  tento  uzel  ze  zásobníku  a  znič  jeho
obal}
   dispose(garbage);
   pom^.node^.left:=VPSStack^.node; {navaž nyní první uzel
                          na zásobníku jako levý podstrom}
   garbage:=Cut; {Vyjmi jej ze zásobníku}
   dispose(garbage); {znič obal}
   SInsert(pom); {vlož obal a uzel operátoru do zásobníku}
 end;
```

8.2 Optimalizace výrazu

Vytvořený syntaktický strom nám dává možnost pracovat algoritmicky se strukturou formule. Strom můžeme procházet principiálně různými způsoby podle toho, co je cílem algoritmu. Existují 4 základní možnosti průchodu pre-order, in-order, post-order a level-order. Anglické slovíčko „order" jasně říká, co se myslí průchodem - jde o pořadí v jakém pracuje s uzly. Na následujících schématech lze pozorovat v jakém pořadí budou danou metodou zpracovány uzly (číslo je pořadí). Pravidlo se vždy aplikuje rekurzivně (s mírnou výjimkou u Level-order) - tedy používá samo sebe na potomky uzlu. Na uzel aplikujeme vybranou operaci (třeba výpis znaku uzlu). Trochu výjimečná je procedura

průchodu Level-order, kde musíme implementovat frontu s ještě nezpracovanými uzly a ty postupně obsluhovat.

```
       1        Pre-order                        7        Post-order
     _/ \_      - operace s uzlem              _/ \_      - Post-order na levý uzel
    2     5     - Pre-order na levý uzel      3     6     - Post-order na pravý uzel
   / \   / \    - Pre-order na pravý uzel    / \   / \    - operace s uzlem
  3   4 6   7                               1   2 4   5

                          4        In-order
                        _/ \_      - Inorder na levý uzel
                       2     6     - operace s uzlem
                      / \   / \    - In-order na pravý uzel
                     1   3 5   7

       1        - úroveň 0   Level-order
     _/ \_                   - operace s uzlem (n-úroveň)
    2     3     - úroveň 1   - vygeneruj uzly nižší n + 1 úrovně a dej do fronty
   / \   / \                 Začni od úrovně 0 - kořene stromu
  4   5 6   7   - úroveň 2   Aplikuj iterativně na všechny uzly ve frontě Level-order
```

Využítí různých způsobu průchodu se dá ilustrovat na následujících příkladech: Výpis výrazu do infixní formy - použijeme In-order, protože chceme mít vždy binární spojky v pořadí operand operátor operand.

Vyhodnocení výrazu - použijeme Post-order, neboť výraz se vyhodnocuje "zdola", tj. než začneme vyhodnocovat spojku, musíme oba podvýrazy již mít vyčíslené

Odstranění negace směrem na proměnné - použijeme Pre-order, jedná se o často prováděnou operaci v logice, chceme aby negace byla jen u logických proměnných, musíme tedy negaci postupně shora od kořene tlačit směrem dolů pomocí logických zákonů jako jsou De-Morganovy zákony, např.

$$\neg(A \wedge B) \Leftrightarrow (\neg A \vee \neg B).$$

Podívejme se blíže na algoritmus pro výpis formule v infixním tvaru čitelném pro uživatele. Rekurzivní procedura typu In-order Dive provede zejména volání sama sebe pro levý podstrom pro logickou spojku, pak se vždy vypíše znak atomu a nakonec se zavolá Dive pro pravý podstrom u spojky. Samozřejmě je třeba vyřešit závorkování (infixní forma, na kterou jsou lidé

zvyklí, se bez závorek neobejde narozdíl od postfixu a stromu). Detaily jsou popsány v komentářích zdrojového kódu.

```
procedure TEditForm.Dive; {vypisuje rekurzivně formuli v
infixním tvaru}
   begin
         if (node^.neg=true) then
           begin   {pokud je uzel negován musíme vložit znak
negace}
             InsertString('~');
             if node^.prior<>6 then
             {ale pokud to není atom jsou nutné závorky}
               begin InsertString('('); end;
           end;
         if (node^.prior <> 6) then
         {není-li atom budeme vypisovat
                        levý podstrom}
           if
(node^.prior>node^.left^.prior)and(not(node^.left^.neg)) then

           {pokud má operátor levého podvýrazu nižší prioritu
musíme
           vložit závorku, jinak by se provedl nejprve uzel,
který právě řešíme}

              begin
                InsertString('(');

                {volá se Dive pro levý podstrom}
                Dive(node^.left); InsertString(')');
              end else Dive(node^.left); {volání bez závorek}
         InsertString(node^.character);              {vložíme
znak spojky uzlu}
         if (node^.prior <> 6) then {není-li atom vypíšeme
pravý podstrom}
           if
((node^.prior>node^.right^.prior)or((node^.prior=2)

and(node^.right^.prior=2)))and(not(node^.right^.neg)) then
```

```
 {pokud má operátor pravého podvýrazu nižší prioritu musíme
vložit závorku, jinak
 by se provedl nejprve uzel, který právě řešíme Pozor! Pro
pravý podstrom je třeba
 závorka, pokud je uzel i jeho pravý podvýraz implikace - není
asociativní}

              begin
                InsertString('(');

                {volá se Dive pro pravý podstrom}
                Dive(node^.right); InsertString(')');
              end
           else Dive(node^.right); {volání bez závorek}
         if (node^.prior<>6)and(node^.neg=true) then
           begin
             InsertString(')');{ukončení závorky pro počáteční
negaci}
           end;
  end;
```

Ve výuce se logika na SŠ většinou vyučuje jen v rámci matematiky a to způsobem, který odpovídá jejímu využití v matematice - tedy především jako formální aparát pro vyjadřování matematických vztahů. Většinou se spokojíme s výukou sémantiky logických spojek, případně s pojmem tautologie, ale existují také jednoduché a čistě formální metody na zjednodušování formulí a případně i určování platnosti, splnitelnosti a ověřování správnosti dedukce. Důležité je, že daná pravidla jsou poměrně jednoduchá a pracují výlučně se syntaxí formule, kterou nyní máme zapsánu velmi vhodně formou stromu. Formální pravidla známe i z jiných oblastí, například z teorie množin, kde třeba známe a často používáme vztah mezi dvěmi množinami a operacemi rozdílu a průniku:

$$A - B = A \cap \overline{B}$$

Podobně lze upravovat nejen výrazy s rovnicemi, ale i logické výrazy. Nejprve se podívejme na vyhodnocení logických konstant.

Pro vyhodnocení logických konstant může využít následující zákony (ekvivalence):

$$A \wedge B \Leftrightarrow B \wedge A, A \vee B \Leftrightarrow B \vee A, A \leftrightarrow B \Leftrightarrow B \leftrightarrow A$$

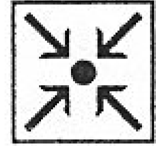
$$\neg 0 \Leftrightarrow 1, \neg 1 \Leftrightarrow 0, A \wedge 0 \Leftrightarrow 0, A \wedge 1 \Leftrightarrow A, A \vee 0 \Leftrightarrow A, A \vee 1 \Leftrightarrow 1, A \leftrightarrow 0 \Leftrightarrow \neg A, A \leftrightarrow 1 \Leftrightarrow A$$

$$A \rightarrow 1 \Leftrightarrow 1, A \rightarrow 0 \Leftrightarrow \neg A, 1 \rightarrow A \Leftrightarrow A, 0 \rightarrow A \Leftrightarrow 1$$

Tyto ekvivalence může zcela přirozeně naimplementovat do systému, který pracuje se syntaktickým stromem a pak už jen stačí napsat algoritmus, který vhodně prochází strom a aplikuje je. Vlastní procházení typu Post-order realizuje procedure RConstEval (uvádíme jen fragment) a každý uzel ja pak testován procedurou , zda se v něm díky logické konstantě nemůže zjednodušovat podle výše uvedených zákonů.

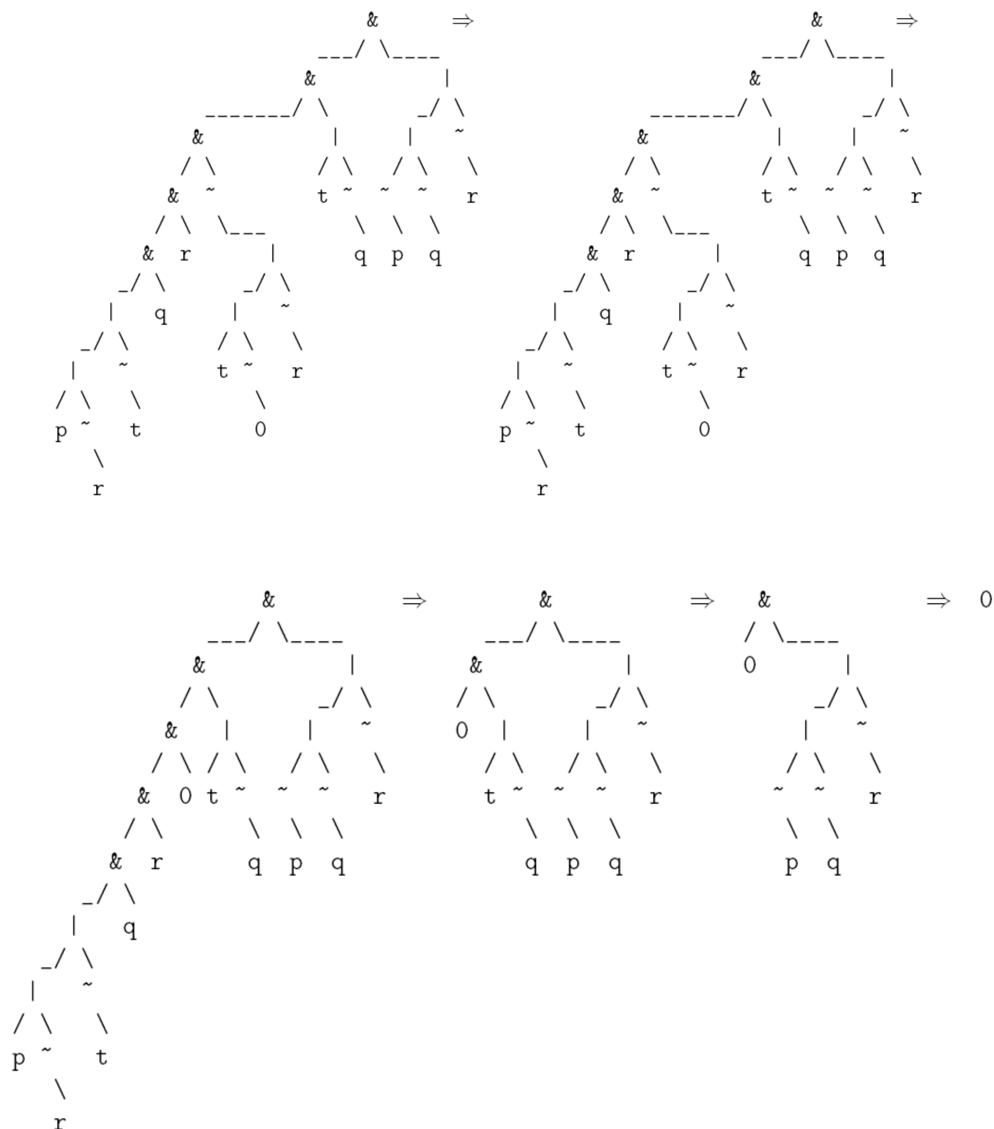
```
procedure TEditForm.RConstEval;
    begin
      if  (tr^.prior <> 6) then
        begin
          RConstEval(tr^.left);           RConstEval(tr^.right);
EvlConst1(tr);
        end; {levý podstrom, pravý podstrom, pak teprve uzel}
    end;
...
procedure TEditForm.EvlConst1;
 {pokus o vyhodnocení logické pravdy nebo nepravdy}
 ...
 begin
 ...
 case pchar of
      '=':begin
            if              pompt2^.character='0'              then
pompt1^.neg:=not(pompt1^.neg);
            p:=pompt1;RdisposeTree(pompt2);
          end;{ekvivalence se vyhodnotí přesně dle ekvivalencí}
```

```
                    { ( A = 1 ) = A
                      ( A = 0 ) = ~A }
 ...
 end;
```



Podívejme se také na příklad takového zjednodušení, které může být velmi účinnou optimalizací a ušetřit nám zbytečné vyhodnocování velké části podmínky. Mějme poměrně složitou podmínku -logický výraz - který obsahuje jednu konstantu 0 (třeba bychom ji dostali po vyhodnocení jednoho porovnání v nějakém programovacím jazyce a nějakém programu v něm vytvořeném).

```
(p|~r|~t)&q&r&~(t|~0|~r)&(t|~q)&(~p|~q|~r)
```



Další zajímavou transformací logického výrazu je převod na tzv. funkčně úplnou množinu spojek. V logice existují kombinace spojek, pomocí kterých

můžeme vyjádřit všechny formule jako ekvivalentní. To by mohlo být zajímavé v případě, že stroj, který pro zpracování našich logických výrazů použijeme, umí velmi rychle oproti jiným spojkám, řešit nějakou konkrétní spojku (procesory počítačů i jiné logické obvody mohou tuto vlastnost mít). Druhou možností jsou například některé důkazové metody (například metoda "reductio ad absurdum", nepřímý důkaz, vyžaduje formule ve formě implikací). Funkčně úplné množiny spojek jsou například:

$$\{\neg, \rightarrow\}, \{\rightarrow\}$$

- negace a implikace, či dokonce pouze implikace (ovšem s pomocí logických konstant)!

$$\{\neg, \wedge, \vee\}, \{\neg, \wedge\}, \{\neg, \vee\}$$

- negace, konjunkce, disjunkce či dokonce negace pouze s jednou těchto spojek; u první množiny lze dostat tvar tzv. negační normální formy - kdy negace se vyskytuje pouze u výrokových proměnný (lze ji vytlačit směrem dolů ve stromě až na listy)

Při převodu výrazu na tyto množiny spojek můžeme využít například tyto zákony (přepis mezi implikací a disjunkcí a konjunkcí, odstranění ekvivalence, vytlačení negace - De Morganovy zákony):

$$A \wedge B \Leftrightarrow \neg(A \rightarrow \neg B), A \vee B \Leftrightarrow \neg A \rightarrow B, A \leftrightarrow B \Leftrightarrow (A \rightarrow B) \wedge (B \rightarrow A), \neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B, \neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B, \neg\neg A \Leftrightarrow A$$

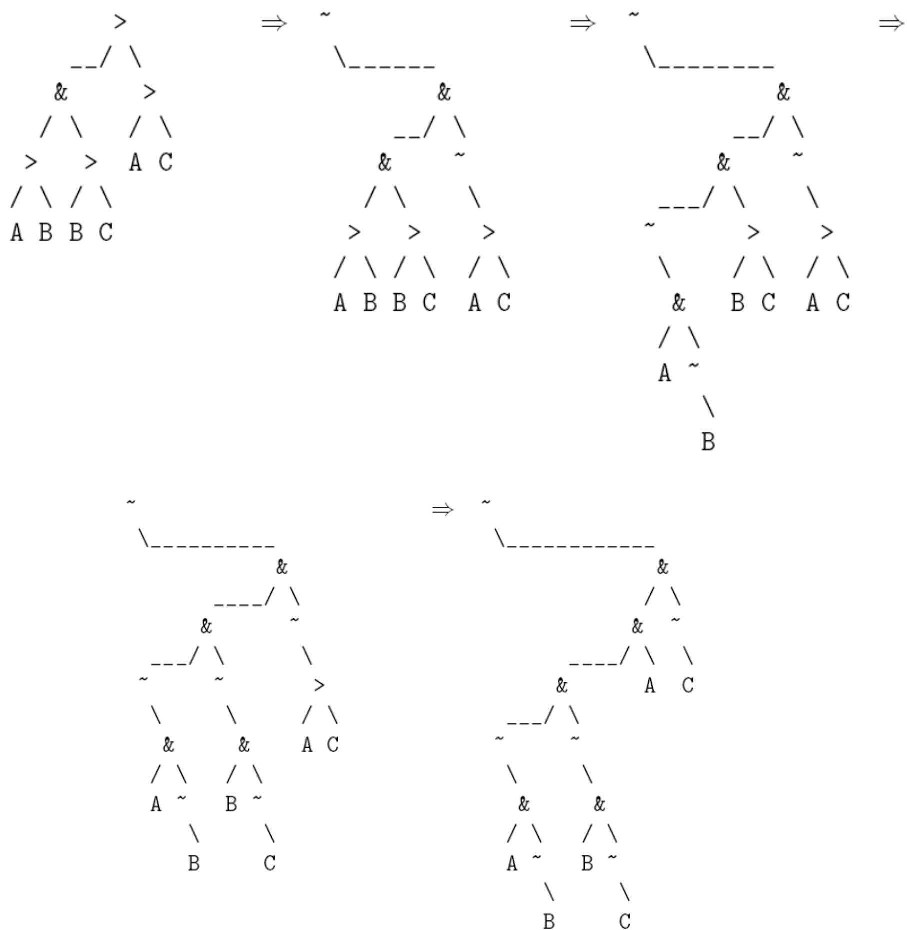
Tyto zákony lze implementovat průchodem typu Pre-order (nejprve se musí změnit spojka na vyšší úrovni a pak se může pokračovat dále na nižší). Podívejme se pro příklad na fragment algoritmu na převod na konjunkce a negace. Procedura Conjunction provede rekurzivně změny spojek a příslušné změny na negované formule dle ekvivalencí.

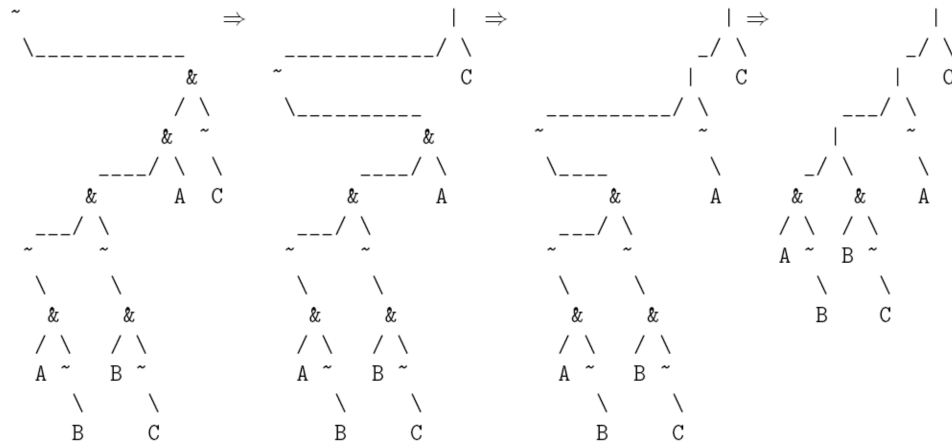
```
procedure TEditForm.Conjunction;{konverze na konjukce a negace}
...
     else if p^.character = '>' then
         begin {implikaci změníme na konjunkci}
           ChangeOper(p,pchar);p^.neg:=not(p^.neg);
           {pozor nová konjukce je negovaná}
           p^.right^.neg:=not(p^.right^.neg);
```

```
        {pozor pravý podstrom je také negovaný}
        ...
      end;
      { ( A > B ) =  ~ ( A & ~ B )  }
   end;
end;
```

A nyní příklad, jak lze nejprve formuli převést pouze na konjunkce, negace a pak na konjunkce, disjunkce a negace, kde negace budou jen na proměnných. Výchozí formule je následující (vlastně jde o známé pravidlo tranzitivity):

```
(((A>B)&(B>C))>(A>C))
```





Nyní si oba převody ještě okomentujeme v infixní formě:

(((A>B)&(B>C))>(A>C))

Implikace na konjunkci:~((A>B)&(B>C)&~(A>C))

Implikace na konjunkci:~(~(A&~B)&(B>C)&~(A>C))

Implikace na konjunkci:~(~(A&~B)&~(B&~C)&~(A>C))

Implikace na konjunkci:~(~(A&~B)&~(B&~C)&A&~C)

~(~(A&~B)&~(B&~C)&A&~C)

Conjunction to disjunction:~(~(A&~B)&~(B&~C)&A)|C

Konjunkce na disjunkci:~(~(A&~B)&~(B&~C))|~A|C

Konjunkce na disjunkci:A&~B|B&~C|~A|C

A&~B|B&~C|~A|C

Nakonec se seznámíme s nejdůležitějším pojmem tzv. konjunktivní normální formy výrazu. Ten nám také umožní dokonce formálně prověřovat dedukci.

Konjunktivní normální forma (KNF) výrazu se skládá z tzv. konjunktů spojených spojkou konjunkce (konečný počet). Konjunkt je tvořen z výrokových proměnných bez nebo s negací spojených výhradně spojkou disjunkce. Tento tvar tedy z hlediska stromu obsahuje směrem od kořene dolů konjunkce a platí, že od první disjunkce se už směrem dolů vyskytují jen disjunkce nebo atomy s negací a bez.

Disjunktivní normální forma (DNF) výrazu je duální ke KNF a skládá se z tzv. disjunktů spojených spojkou disjunkce (konečný počet). Disjunkt je tvořen z výrokových proměnných bez nebo s negací spojených výhradně spojkou konjunkce. Tento tvar tedy z hlediska stromu obsahuje směrem od kořene dolů disjunkce a platí, že od první konjunkce se už směrem dolů vyskytují jen konjunkce nebo atomy s negací a bez. Příkladem DNF je vygenerovaná formule z minulého příkladu.

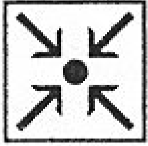
A&~B|B&~C|~A|C

KNF se používá především v oblasti automatizovaného dokazování při použití klauzulární rezoluční metody a DNF se dá využít pro zjištění platnosti formule (zda je tautologie) - DNF tautologie by po zjednodušení měla degradovat na 1 (pravdu). DNF je navíc velmi expresivní, pokud chceme určit, za jakých pravdivostních hodnot výchozích proměnných je výraz pravdivý. Druhá stránka věci je také pro praxi velice užitečná, neboť jednoduchými pravidly lze DNF i KNF minimalizovat, což může významně zjednodušit a zmenšit formuli (samozřejmě menší formule, znamená mnohem méně vyhodnocování při výpočtu výrazu). Nejprve si projděme pravidla pro přepis do normálních forem pomocí ekvivalencí (existují samozřejmě i sémantické metody, kdy z tabulky můžeme vytvořit tzv. úplné normální formy). Ekvivalence, kromě již výše použitých pravidel, zahrnují především dvě pravidla, která umožňují změnit hierarchii negace mezi spojkami konjunkce a disjunkce (jde o distributivní zákon), dále zákony absorpce, absorpce negace, idempotence a komplementarita, a nakonec rozšíření.

$$A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C), A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C),$$
$$A \vee (A \wedge B) \Leftrightarrow A, A \wedge (A \vee B) \Leftrightarrow A, A \vee (\neg A \wedge B) \Leftrightarrow A \vee B, \neg A \vee (A \wedge B) \Leftrightarrow$$
$$\neg A \vee B, A \wedge (\neg A \vee B) \Leftrightarrow A \wedge B, \neg A \wedge (A \vee B) \Leftrightarrow \neg A \wedge B,$$
$$A \vee A \Leftrightarrow A, A \wedge A \Leftrightarrow A, A \vee \neg A \Leftrightarrow 1, A \wedge \neg A \Leftrightarrow 0,$$
$$(A \wedge \neg B) \vee (A \vee B) \Leftrightarrow A$$

Pozn. V automaticky vyhodnocených příkladech se používá v algoritmech někdy pro jednodušší manipulaci (např. u zákona rozšíření) přiřazení logických konstant za A a B, tak aby výsledek po vyhodnocení odpovídal ekvivalenci.

Příklad 2.



Mějme formuli vyjadřující, že existuje k implikaci i její obrácená implikace, pak lze odvodit i ekvivalenci obou formulí, které jsou argumenty implikace.

(A>B)>((B>A)>(A=B))

Nejprve se podívejme, které operace musíme udělat, abychom dospěli k DNF. Na této DNF je vidět, za jakých podmínek je platná. Obsahuje 4 disjunkty a můžeme z nich i vyčíst, při jakých hodnotách dosazených za logické proměnné bude výraz pravdivý. Disjunkt 1 říká, že výraz platí, pokud A platí (tedy A = 1) a zároveň platí ŘB (tedy B = 0), disjunkt 2 - A = 0, B = 1, disjunkt 3 - A = 0, B = 0, disjunkt 4 - A = 1, B = 1. V tomto případě jednoduchou úvahou vidíme, že v podstatě to jsou všechny možnosti, jaké hodnoty mohou být A a B dosazeny a tudíž formule je tautologie. Minimalizace nám však toto zajistí algoritmicky.

(A>B)>((B>A)>(A=B))
Implikace na konjunkci:~((A>B)&~(B>A>(A=B)))
Konjunkce na disjunkci:~(A>B)|(B>A>(A=B))
Implikace na konjunkci:A&~B|(B>A>(A=B))
Implikace na konjunkci:A&~B|~((B>A)&~(A=B))
Konjunkce na disjunkci:A&~B|~(B>A)|(A=B)
Implikace na konjunkci:A&~B|B&~A|(A=B)
Ekvivalence na konjunkci:A&~B|B&~A|~(A&~B)&~(B&~A)
Konjunkce na disjunkci:A&~B|B&~A|(~A|B)&~(B&~A)
Konjunkce na disjunkci:A&~B|B&~A|(~A|B)&(~B|A)
Distribuce:A&~B|~A&B|~A&(~B|A)|B&(~B|A)
Distribuce:A&~B|~A&B|~B&~A|A&~A|B&(~B|A)
Komplementarita:A&~B|~A&B|~A&~B|0&~A|B&(~B|A)
Distribuce:A&~B|~A&B|~A&~B|0|~B&B|A&B
Komplementarita:A&~B|~A&B|~A&~B|0|0&B|A&B

DNF: A&~B|~A&B|~A&~B|A&B

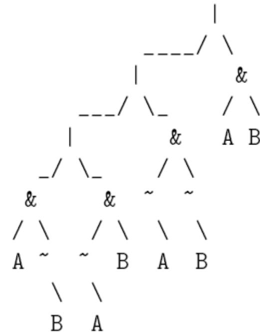
Nebo výstižněji pomocí stromu.

```
   >        ⇒  ~           ⇒      |      ⇒     |        ⇒      |        ⇒
  / \__         \__            __/ \__       _/ \__          _/ \
 >     >           &          ~       >     &      >        &   ~
/ \   / \         / \          \     / \   / \    / \      / \   \__
A B  >   =       >  ~           >   >   =  A ~   >   =     A ~      &
    / \ / \     / \  \__       / \ / \ / \    \ / \ / \      \     / \
    B A A B     A B     >      A B B A A B    B B A A B      B    >  ~
                       / \                                       / \  \
                      >   =                                      B A   =
                     / \ / \                                          / \
                     B A A B                                          A B
```

```
          |  ⇒           |  ⇒           |        ⇒           |        ⇒
      ___/ \         ___/ \         ___/ \____           ___/ \___
     |      =       |      =       |          &         |         &
    _/ \   / \     _/ \   / \     _/ \    ___/ \       _/ \      / \
  &   ~    A B    &    &   A B   &    &   ~     ~     &    &    |  ~
 / \   \         / \  / \       / \  / \   \     \   / \  / \  / \  \
 A ~    >        A ~  B ~       A ~  B ~    &     &  A ~  B ~  ~  B  &
    \  / \          \    \         \    \  / \   / \    \    \  \   / \
    B  B A          B    A         B    A  A ~   B ~    B    A  A   B ~
                                             \     \                   \
                                             B     A                   A
```

```
           |        ⇒                   |   ⇒                     | ⇒
       ___/ \___                  ______/ \                    ___/ \
      |         &               |         &                   |      &
     _/ \      / \_          ___/ \_     / \_            ____/ \    / \_
   &    &     |    |        |       &     B   |          |       &  B   |
  / \  / \   / \  / \     _/ \_    / \_     / \      ___/ \_    / \    / \
  A ~  B ~  ~  B ~  A    &     &  ~    |    ~  A    |       &  A ~    ~  A
     \    \  \    \     / \   / \  \  / \    \    _/ \_    / \    \    \
     B    A  A    B     A ~  ~  B  A ~  A    B  &     &  ~  ~     A    B
                          \  \      \          / \   / \  \  \
                          B  A      B          A ~  ~  B  B  A
                                                  \  \
                                                  B  A
```

```
                   |  ⇒                      |⇒                      |⇒
               ___/ \                     ___/ \                  __/ \
              |      &                   |      &                |     &
         ____/ \    / \_                / \_   / \              / \   / \
        |       &  B   |               |   &  A B              |  &   A B
    ___/ \_    / \    / \         ____/ \ / \              ____/ \ / \
    |      &  O ~    ~  A        |      O ~  B            |      O O B
  _/ \_   / \    \    \       ___/ \_      \          ___/ \_
 &    &  ~  ~    A    B      |      &      B          |      &
/ \  / \  \  \            _/ \_    / \             _/ \_    / \
A ~  ~  B  A  B          &     &  ~  ~            &     &  ~  ~
   \  \                 / \   / \  \  \          / \   / \  \  \
   B  A                 A ~  ~  B  A  B          A ~  ~  B  A  B
                           \  \                     \  \
                           B  A                     B  A
```



Nyní můžeme zkusit aplikovat pravidla pro minimalizaci formule a dostaneme logickou konstantu 1. Tudíž formule je opravdu logicky platná (tautologie - zákon). Takovou podmínku bychom bez problému mohli zcela vynechat a považovat ji za vždy splněnou.

A&~B|~A&B|~A&~B|A&B

Rozšíření:0&~B|~A&B|1&~B|A&B
Rozšíření:0&~B|0&B|1&~B|1&B
Rozšíření:0|1
Minimální DNF: 1

Příklad:

Na dalším příkladě můžeme vidět, jak drasticky se zjednoduší pomocí minimalizace formule - tedy ušetříme mnoho potenciálních vyhodnocení. Mějme následující formuli:

(~p>(q&~r))>(~q|r)

Nejprve ji po krocích převedeme na DNF.
Implikace na konjunkci:~((~p>q&~r)&~(~q|r))
Konjunkce na disjunkci:~(~p>q&~r)|~q|r

Implikace na konjunkci:~p&~(q&~r)|~q|r

Konjunkce na disjunkci:~p&(~q|r)|~q|r

Distribuce:~q&~p|r&~p|~q|r

DNF: ~p&~q|~p&r|~q|r

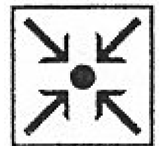
Minimalizací dostaneme překvapivě jednoduchou formuli a to pouze aplikací dvou absorpcí.

Absorpce:0&~q|~p&r|~q|r

Absorpce:0&~q|0&r|~q|r

Minimální DNF: ~q|r

Posledním použitím, které sice úzce nesouvisí s cílem našeho článku, je možnost minimalizací DNF zjišťovat, zda závěr vyplývá z daných předpokladů. Ukažme si to pouze na příkladu.



Příklad 3.

Nechť jsou daná tři tvrzení - předpoklady:

1. Jan je učitel.
2. Neplatí, že Jan je učitel a zároveň je bohatý.
3. Je-li Jan rockový zpěvák, pak je bohatý.

Chtěli bychom prověřit závěr - Z. Jan není rockový zpěvák.

Nejprve musíme zvolit logické proměnné.

u - Jan je učitel.

b - Jan je bohatý.

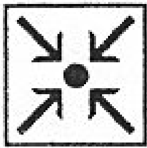
z - Jan je rockový zpěvák.

Nyní musíme sestrojit výrazy pomocí spojek pro tvrzení 1., 2., 3. a spojit je všechny konjunkcí (platí současně), tuto konjunkci pak dáme jako první podvýraz implikace a druhým bude závěr Z. (Implikace vyjadřuje, že závěr vyplývá z předpokladu). Formuli následně převedeme na minimální DNF a vyjde-li 1, pak se jedná o správnou dedukci (systém Bachelor toto přímo umožňuje).

u&~(u&b)&(z>b)>~z

Implikace na konjunkci:~(u&~(u&b)&(z>b)&z)
Konjunkce na disjunkci:~(u&~(u&b)&(z>b))|~z
Konjunkce na disjunkci:~(u&~(u&b))|~(z>b)|~z
Konjunkce na disjunkci:~u|u&b|~(z>b)|~z
Implikace na konjunkci:~u|u&b|z&~b|~z
Absorpce negace:~u|b&1|~b&z|~z
Absorpce negace:~u|b&1|~b&1|~z
Rozšíření:~u|0|1|~z
Výraz je platný. {Dedukce je správná}

Druhý zajímavý příklad umožňuje pomocí DNF zjistit, kdy dává sada výrazů smysl - v tomto případě tak chceme odhalit viníka trestného činu (předpokládáme, že mluví pravdu).



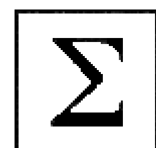
Příklad 4.

Brown, Jones a Smith jsou podezřelí z podvodu. Svědčili pod přísahou takto:
1. Brown: Jones je vinen a Smith je nevinen.
2. Jones: Je-li vinen Brown, pak je vinen i Smith.
3. Smith: Já jsem nevinen, ale alespoň jeden ze zbývajících obviněných je vinen.
Nejprve musíme zvolit logické proměnné.
u - Brown je vinen.
b - Jones je vinen.
z - Smith je vinen.
Nyní musíme sestrojit výrazy pomocí spojek pro tvrzení 1., 2., 3. a spojit je všechny konjunkcí (platí současně). Výraz následně převedeme na minimální DNF a disjunkty nám ukáží informace, kdy je konjunkce tvrzení pravdivá -

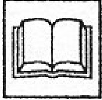
dává smysl a také z nich přímo vyčteme, jaká musí vina a nevina jednotlivých obviněných.

(J&~S)&(B>S)&(~S&(J|B))
Implikace na konjunkci:J&~S&~(B&~S)&~S&(J|B)
Konjunkce na disjunkci:J&~S&(~B|S)&~S&(J|B)
Distribuce:~B&J&~S&~S&(J|B)|S&J&~S&~S&(J|B)
Distribuce:J&J&~S&~S&~B|B&J&~S&~S&~B|S&J&~S&~S&(J|B)
Idempotence:~B&1&J&~S&~S|B&J&~S&~S&~B|S&J&~S&~S&(J|B)
Idempotence:~B&1&J&1&~S|B&J&~S&~S&~B|S&J&~S&~S&(J|B)
Komplementarita:~B&1&1&J&~S|0&~B&J&~S&~S|S&J&~S&~S&(J|B)
Idempotence:~B&1&1&J&~S|0&~B&J&1&~S|S&J&~S&~S&(J|B)
Distribuce:~B&J&~S|0|J&J&~S&~S&S|B&J&~S&~S&S
Idempotence:~B&J&~S|0|1&J&~S&~S&S|B&J&~S&~S&S
Idempotence:~B&J&~S|0|1&J&1&~S&S|B&J&~S&~S&S
Komplementarita:~B&J&~S|0|1&J&1&0&S|B&J&~S&~S&S
Idempotence:~B&J&~S|0|1&0&1&J&S|B&J&1&~S&S
Komplementarita:~B&J&~S|0|1&0&1&J&S|B&J&1&0&S
Formule je konzistentní.Její modely vyjadřuje DNF:
~B&J&~S {Tedy vinen je Jones a ostatní jsou nevinní.}

- Význam dedukce pro praxi
- Formální vs. Sémantická dedukce



9 Literatura



[Be03] BENEŠ, M. Překladače. Učební text VŠB-TUO:Ostrava, 2003, Dokument dostupný (2004): http://www.cs.vsb.cz/benes/vyuka/udp/

[Ce92] ČEŠKA, Milan, RÁBOVÁ, Zdena. Gramatiky a jazyky. Brno, VUT 1992.Dokument dostupný na URL (2005): http://www.fit.vutbr.cz/study/courses/TI1/public/Texty/ti.pdf

[Dv92] DVOŘAK, Stanislav. Dekompozice a rekurzivní algoritmy. Grada 1992, Praha

[Ch84] CHYTIL, Milan. Automaty a gramatiky. Praha, SNTL 1984.

[Ha03] HABIBALLA, H. Regulární a bezkontextové jazyky I. Ostrava : Ostravská Univerzita, 2003. 140 s.

[Ha05] HABIBALLA, H. Regulární a bezkontextové jazyky II. Ostrava : Ostravská Univerzita, 2005.

[Habiballa, 2009a] Habiballa, H. Bachelor – propositional logic rewrite systém. Dostupné z: http://www1.osu.cz/home/habibal/page9.html

[Ho79] HOPCROFT, J.E., ULLMAN, J. D. Introduction to Automata theory, Languages and Computation. Addison-Wesley, Reading (Mass.), 1979

[Ka02] KASTENS, U.: Demonstration of parsing methods, http://www.uni-paderborn.de\fachbereich\AG\agkastens\compiler\parsdemo\index.html

[Le02] LEWIS, F.D. Recursive Descent Parsing, http://cs.engr.uky.edu\~lewis\essays\compilers\rec-des.html

[Ja97] JANČAR, Petr. Teorie jazyků a automatů. VŠB TU Ostrava, Dokument dostupný na URL: http://www.cs.vsb.cz/jancar/ (2005)

[Ji88] JINOCH, J., MULLER, K., VOGEL, J. Programování v jazyce Pascal. SNTL 1988, Praha

[No05] NOHÁČEK, J. Automatizace tvorby překladačů. Diplomová práce OU:2005, Ostrava.

Literatura

Základní:

Češka, M., Hruška, J., Beneš. M. Překladače, VUT Brno, 1993 dokument dostupný na: www.fit.vutbr.cz/~meduna/fjp/skripta.pdf (2014).

Doporučená:

L. MOLNÁR, M. ČEŠKA, B. MELICHAR Gramatiky a jazyky, ALFA/SNTL 1987.

Melichar, Bořivoj. Konstrukce překladačů. 1.a 2. část. Praha : ČVUT, 1999.

Rozšiřující:

AHO, Alfred V., Ravi SETHI a Jeffrey D. ULLMAN. Compilers, principles, techniques, and tools. Reading: Addison-Wesley Publishing Company, 1987. x, 796 s. ISBN 0-201-10088-6.

APPEL, Andrew W. Modern compiler implementation in Java. Cambridge: Cambridge University Press, 1998. x, 548 s. ISBN 0-521-58388-8.

COOPER, Keith D. a Linda TORCZON. Engineering a compiler. San Francisco: Morgan Kaufmann Publishers, 2004. xxx, 801 s. ISBN 1-55860-698-X.

GRUNE, Dick. Modern compiler design. Chichester: John Wiley & Sons, 2000. xviii, 736. ISBN 0-471-97697-0.

Š. Vavrečková. Překladače, SLU Opava, 2006.