Dr. Ing. Miroslav Beneš

# PŘEKLADAČE

# Obsah

**1 Základní pojmy** **1**
1.1 Úvod . . . . . 1
1.1.1 Vývoj technik strojového překladu . . . . . 1
1.1.2 Přístupy ke strojovému překladu . . . . . 2
1.1.3 Další použití překladačů . . . . . 5
1.2 Struktura překladače . . . . . 6
1.2.1 Lexikální analýza . . . . . 7
1.2.2 Syntaktický analyzátor . . . . . 8
1.2.3 Sémantická analýza . . . . . 9
1.2.4 Generování mezikódu . . . . . 9
1.2.5 Optimalizace kódu . . . . . 10
1.2.6 Generování cílového kódu . . . . . 11
1.2.7 Tabulka symbolů . . . . . 11
1.2.8 Diagnostika a protokol o průběhu překladu . . . . . 12
1.3 Organizace překladu . . . . . 13
1.3.1 Fáze překladu . . . . . 13
1.3.2 Průchody . . . . . 14
1.4 Příbuzné programy . . . . . 14
1.5 Automatizace výstavby překladačů . . . . . 16

**2 Lexikální analýza** **19**
2.1 Činnost lexikálního analyzátoru . . . . . 19
2.2 Základní pojmy . . . . . 20
2.2.1 Symboly, vzory, lexémy . . . . . 20
2.2.2 Atributy symbolů . . . . . 21
2.3 Vstup zdrojového textu . . . . . 22
2.4 Specifikace a rozpoznávání symbolů . . . . . 24
2.4.1 Regulární výrazy . . . . . 25
2.4.2 Regulární definice . . . . . 26
2.4.3 Konečné automaty . . . . . 27
2.5 Implementace lexikálního analyzátoru . . . . . 27
2.5.1 Přímá implementace . . . . . 28
2.5.2 Implementace lexikálního analyzátoru jako automatu se stavovým řízením 30
2.6 Lex — generátor lexikálních analyzátorů . . . . . 32
2.6.1 Činnost programu `lex` . . . . . 32
2.6.2 Struktura zdrojového textu . . . . . 33

2.6.3 Zápis regulárních výrazů . . . . . 34
2.6.4 Komunikace s okolím . . . . . 34
2.7 Zotavení po chybě v lexikální analýze . . . . . 36

**3 Syntaktická analýza 39**
3.1 Činnost syntaktického analyzátoru . . . . . 39
3.2 Syntaktická analýza shora dolů . . . . . 39
3.2.1 Množiny FIRST a FOLLOW . . . . . 40
3.2.2 Konstrukce rozkladových tabulek . . . . . 41
3.2.3 LL(1) gramatiky . . . . . 43
3.2.4 Transformace na LL(1) gramatiku . . . . . 44
3.2.5 Analýza rekurzivním sestupem . . . . . 45
3.2.6 Nerekurzivní prediktivní analýza . . . . . 48
3.2.7 Zotavení po chybě při analýze shora dolů . . . . . 49

**4 Syntaxí řízený překlad 55**
4.1 Základní pojmy teorie překladu . . . . . 55
4.2 Atributovaný překlad . . . . . 57
4.2.1 Atributové překladové gramatiky . . . . . 58
4.2.2 Graf závislosti . . . . . 61
4.2.3 Pořadí vyhodnocení pravidel . . . . . 62
4.3 Vyhodnocení S-atributových definic zdola nahoru . . . . . 63
4.4 L-atributové definice . . . . . 67
4.5 Překlad shora dolů . . . . . 68
4.5.1 Odstranění levé rekurze z překladového schematu . . . . . 68
4.5.2 Implementace prediktivního syntaxí řízeného překladače . . . . . 69
4.6 Vyhodnocení dědičných atributů zdola nahoru . . . . . 70

**5 Tabulka symbolů 73**
5.1 Informace v tabulce symbolů . . . . . 73
5.2 Organizace tabulky symbolů . . . . . 76
5.2.1 Operace nad tabulkou symbolů . . . . . 76
5.2.2 Implementace tabulek pro jazyky bez blokové struktury . . . . . 76
5.2.3 Implementace blokově strukturované tabulky symbolů . . . . . 77

**6 Struktura programu v době běhu 81**
6.1 Podprogramy . . . . . 81
6.1.1 Statická a dynamická struktura podprogramů . . . . . 81
6.2 Organizace paměti . . . . . 83
6.3 Strategie přidělování paměti . . . . . 84
6.3.1 Statické přidělování . . . . . 85
6.3.2 Přidělování na zásobníku . . . . . 85
6.3.3 Přidělování z hromady . . . . . 86
6.4 Metody přístupu k nelokálním objektům . . . . . 86
6.5 Předávání parametrů do podprogramů . . . . . 88
6.5.1 Předávání parametrů hodnotou a výsledkem . . . . . 88
6.5.2 Předávání parametrů odkazem . . . . . 89



6.5.3 Předávání parametrů jménem . . . . . 89
6.5.4 Předávání procedur a funkcí . . . . . 90

**7 Typová kontrola 91**
7.1 Typové systémy . . . . . 92
7.1.1 Typové výrazy . . . . . 92
7.1.2 Statická a dynamická kontrola typů . . . . . 96
7.1.3 Zotavení po chybě při typové kontrole . . . . . 96
7.2 Ekvivalence typových výrazů . . . . . 96
7.3 Typové konverze . . . . . 98
7.4 Přetěžování funkcí a operátorů . . . . . 98
7.5 Polymorfické procedury a funkce . . . . . 99
7.5.1 Unifikace typových výrazů . . . . . 100

**8 Generování intermediárního kódu 101**
8.1 Intermediární jazyky . . . . . 101
8.1.1 Grafová reprezentace . . . . . 101
8.1.2 Zásobníkový kód . . . . . 102
8.1.3 Tříadresový kód . . . . . 104
8.2 Deklarace . . . . . 106
8.2.1 Deklarace proměnných . . . . . 106
8.2.2 Deklarace v jazycích s blokovou strukturou . . . . . 107
8.3 Přiřazovací příkazy a výrazy . . . . . 108
8.3.1 Přidělování dočasných proměnných . . . . . 109
8.3.2 Adresování prvků polí . . . . . 109
8.3.3 Konverze typů během přiřazení . . . . . 112
8.4 Booleovské výrazy . . . . . 113
8.4.1 Reprezentace booleovských výrazů číselnou hodnotou . . . . . 114
8.4.2 Zkrácené vyhodnocování booleovských výrazů . . . . . 114
8.5 Příkazy pro změnu toku řízení . . . . . 116
8.6 Selektivní příkazy . . . . . 118
8.7 Backpatching . . . . . 119
8.7.1 Booleovské výrazy . . . . . 120
8.7.2 Překlad řídicích příkazů . . . . . 122
8.7.3 Volání podprogramů . . . . . 124

# Kapitola 1

# Základní pojmy

V této kapitole se budeme zabývat obecným popisem činnosti a struktury překladače, jeho komunikace s okolím a některými podpůrnými prostředky používanými při výstavbě překladačů.

## 1.1 Úvod

*Překladač* je obvykle program, který čte *zdrojový program* (source program) a převádí ho do ekvivalentního *cílového programu* (object program). Zdrojový program je napsaný ve *zdrojovém jazyce*, cílový program je v *cílovém jazyce*. Důležitou částí tohoto procesu překladu jsou *diagnostické zprávy*, kterými překladač informuje uživatele například o přítomnosti chyb ve zdrojovém programu. Techniky překladačů se používají i pro realizaci počítačových architektur specializovaných na vyšší programovací jazyky (Modula, Lisp, Prolog). V tomto učebním textu ale budeme pojem překladač používat pouze pro program. Typickým zdrojovými jazyky budou programovací jazyky jako Modula-2, Pascal nebo C; typickým cílovým jazykem pro nás bude strojový kód nebo jazyk asembleru nějakého počítače.

### 1.1.1 Vývoj technik strojového překladu

První počítače byly velmi jednoduché, například počítač Mark 1 z roku 1948 měl pouze sedm instrukcí a 32 slov hlavní paměti. Pro takový počítač postačovalo vkládání programů pomocí posloupností binárních číslic. S příchodem složitějších počítačů se rozšiřovaly také instrukční soubory a koncem 40. let bylo poukázáno na to, že převod mnemonických názvů instrukcí do binárního kódu může být proveden pomocí počítače. Programy, které to prováděly, se nazývaly *asemblery* a příslušný mnemotechnický kód *jazyk asembleru*.

Další krok spočíval v zavedení *autokódů*, které umožňovaly reprezentovat jednou instrukcí několik strojových operací. Programy zajišťující jejich překlad se již nazývaly *překladače*, nebo také *kompilátory*. Jedním z používaných autokódů byl například MAT pro počítač Minsk-22, jehož mnemotechnické kódy byly odvozené z českých názvů operací.

Pojem *překladač* se používá od začátku 50. let, kdy se začaly vyvíjet uživatelsky orientované programovací jazyky vyšší úrovně, podstatně méně závislé na strojovém kódu konkrétního počítače. V tu dobu však ještě vládla všeobecná skepse nad použitelností "automatického programování," jak se tehdy programování ve vyšších jazycích nazývalo. První jazyky tohoto typu (např. FORTRAN) a autokódy, ze kterých se vyvinuly, však byly silně poznamenány





tehdy existujícími instrukčními soubory počítačů. Například FORTRAN IV umožňoval práci pouze s trojrozměrnými poli, neboť jeho první implementace byla provedena na počítači IBM 709, který měl pouze tři indexové registry. Dokonce i jazyk C, který se objevil uprostřed 70. let, má některé konstrukce (např. operátor inkrementace ++) zavedené díky dostupnosti ekvivalentních instrukcí původního cílového počítače PDP-11.

Algol 60, navržený ve skutečnosti již v roce 1958, přinesl další nový přístup. Byl navržen s ohledem na řešení konkrétních problémů a potlačoval otázky týkající se možností překladu na konkrétních počítačích. Umožňoval například užití lokálních proměnných a rekurzivních volání procedur. Již se nezabýval tím, jak provést překlad na počítači s jediným společným adresovým prostorem a jedinou instrukcí skoku do podprogramu. Tento přístup je v moderních programovacích jazycích běžný. Jazyky jako Pascal, Modula-2 a Ada byly navrženy nezávisle na jakékoliv konkrétní architektuře.

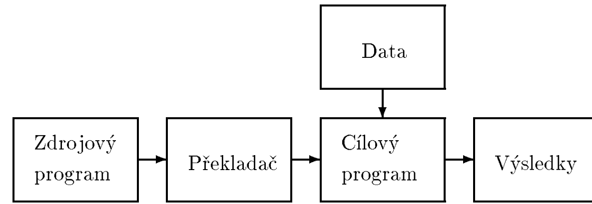
Moderní jazyky vysoké úrovně svým obvykle stručným zápisem umožňují zvýšit produktivitu práce programátora, poskytují různá sémantická omezení (např. typovou kontrolu), kterými se dají redukovat logické chyby v programech, a zjednodušují ladění programů. Další velmi významnou vlastností současných programovacích jazyků je možnost vytváření strojově nezávislých programů, které se dají přenášet i mezi principiálně různými architekturami počítačů. Jejich nevýhodou je rychlost překladu (typicky 2–10 krát nižší než u ručně psaných programů v jazyce asembleru) a velikost, jak překladače, tak přeloženého kódu. Tyto nevýhody jsou však redukovány s rozvojem moderních počítačových architektur. V oblasti návrhu a implementace jazyků se nyní často dostáváme do zcela opačné situace, než jaká byla na počátku vývoje jazyků, kdy jsou navrhovány procesory již s ohledem na překlad konkrétních jazyků (existují například specializované procesory pro Lisp, Pascal nebo Modulu).

Teorie překladu a formálních jazyků dnes umožňuje běžně používané jazyky překládat bez obtíží. Pro automatickou výstavbu překladačů je k dispozici mnoho specializovaných prostředků. Zatímco na vývoj prvního překladače jazyka FORTRAN bylo třeba 18 "člověkoroků," nyní je vytvoření jednoduchého překladače jazyka Pascal zvládnutelné i pro studenta vysoké školy.

### 1.1.2 Přístupy ke strojovému překladu

Máme-li program napsaný v některém vyšším programovacím jazyce, existuje několik možných přístupů k jeho spuštění. Buď můžeme program převést do ekvivalentního programu ve strojovém kódu počítače — překladače tohoto typu se označují názvem *kompilátory* nebo *kompilační překladače*, nebo můžeme napsat program, který bude interpretovat příkazy zdrojového jazyka tak, jak jsou napsané, a přímo provádět odpovídající akce. Programy realizující druhý přístup se nazývají *interprety* nebo *interpretační překladače*. Obrázky 1.1 a 1.2 představují schémata činnosti obou typů překladačů.

Výhodou kompilace je, že analýza zdrojového programu a jeho překlad se provádějí jen jednou, i když může jít o časově dosti náročný proces. Dále již spouštíme pouze ekvivalentní program ve strojovém kódu, který je výsledkem překladu. Nevýhodou je někdy dosti obtížné hledání chyb ve zdrojovém programu, pokud máme pouze informace o místu chyby vyjádřené v pojmech strojového jazyka (adresy, výpisy obrazu paměti). Moderní překladače však často vytvářejí zároveň s cílovým kódem i pomocné datové struktury, které umožňují provádět ladění programu přímo na úrovni zdrojového jazyka — provádět program po jednotlivých příkazech, vypisovat hodnoty proměnných nebo posloupnost volání funkcí a hodnot jejich parametrů.




Obrázek 1.1: Kompilační překladač

Kód generovaný kompilačním překladačem nemusí být obecně ekvivalentní se strojovým kódem nějakého konkrétního počítače. Obecně můžeme cílové kódy podle jejich vztahu k určitému procesoru a operačnímu systému rozdělit takto:

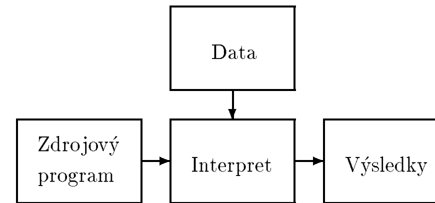
- *Čistý strojový kód.* Jedná se o strojový kód konkrétního počítače bez předpokladu existence určitého operačního systému nebo knihoven. Čistý strojový kód obsahuje pouze instrukce z instrukčního souboru počítače, pro který jsou překládané programy určeny. Tento přístup je velmi řídký, občas se používá pro jazyky určené k vytváření systémových programů (např. jader operačních systémů, které pracují autonomně bez další programové podpory).
- *Rozšířený strojový kód.* Tento typ zahrnuje kromě instrukcí daných architekturou procesoru také podprogramy operačního systému a podpůrné knihovní podprogramy (např. pro matematické funkce). Rozšířený strojový kód se dá považovat za kód *virtuálního počítače*, tvořeného kombinací konkrétního technického a programového vybavení nějakého počítače. Poměr obou složek se může u konkrétních implementací lišit, například překladač jazyka FORTRAN obvykle využívá knihoven pouze pro vstupy a výstupy a pro matematické funkce, zatímco velká část moderních překladačů pracuje s operacemi pro bitová pole, volací posloupnosti procedur a funkcí nebo pro dynamické přidělování paměti.
- *Virtuální strojový kód.* Nejobecnější forma strojového kódu obsahuje pouze virtuální instrukce, které nejsou závislé na žádné konkrétní architektuře nebo konkrétním operačním systému. Tato forma umožňuje vytvářet *přenositelné překladače*; při přenosu stačí pouze napsat interpret virtuálního kódu. Příkladem takového překladače je Wirthův Pascal P, jehož výstupem je tzv. *P-kód* pro virtuální zásobníkový počítač. Velice rychlá přenositelnost takového překladače možná byla jedním z důvodů velké popularity Pascalu.

Další vlastností cílového kódu, který podstatně ovlivňuje složitost návrhu kompilačního překladače, je jeho formát. Pro cílový kód se nejčastěji používá jeden z následujících formátů:

- *Symbolický formát.* Cílový program v symbolickém formátu má obvykle tvar zdrojového souboru v jazyce asembleru. V tomto případě je značně ulehčena práce překladače, neboť se nemusí zabývat například řešením dopředných odkazů v programu nebo přidělováním adres pro data. Tento přístup je častý pod operačním systémem Unix a



je vhodný zejména tam, kde chceme překladač využívat k vytváření programů pro jiný počítač, než na kterém překladač běží (tzv. *křížový překladač*). I přes uvedené výhody se však nedoporučuje, protože se tak silně zpomaluje překlad (je třeba provést konverzi vnitřních datových struktur na text a ten musí zase asembler znovu analyzovat). Pro účely kontroly vygenerovaného kódu je však vhodné, když překladač dovede kód vypsat v symbolickém tvaru.
- *Relokatibilní binární formát.* Tento formát obsahuje cílový kód v binárním tvaru, ovšem bez vyřešených odkazů na externí symboly a pouze s adresami, počítanými relativně od začátku nějakého stanoveného úseku. Takový tvar je typický pro výstup z asembleru, takže se ušetří jeden krok následného zpracování cílového kódu. Symbolický a relokatibilní binární formát umožňují modulární překlad, odkazy na moduly překládané z jiných jazyků a využívání podpůrných knihoven podprogramů. K tomu však vyžadují dodatečné zpracování spojovacím programem.
- *Absolutní binární formát (Load-and-Go).* Program v absolutním binárním tvaru je překladačem ihned po překladu spuštěn. Tím se obejde pomalá fáze sestavování spustitelného programu za cenu omezené dostupnosti vazeb na externí knihovny. Navíc je pro každé spuštění programu nutný jeho opětovný překlad. Tento přístup je výhodný pro studentské programy a pro ladění, kdy se předpokládá častější překlad než spouštění programů.




Obrázek 1.2: Interpretační překladač

Interpretace je mnohem pomalejší než kompilace, neboť je třeba analyzovat zdrojový příkaz pokaždé, když na něj program narazí. Pro poměr mezi rychlostí interpretovaného a kompilovaného programu se uvádějí hodnoty mezi 10:1 až 100:1, v závislosti na konkrétním jazyce. Interprety bývají také náročné na paměťový prostor, neboť i při běhu programu musí být stále k dispozici celý překladač.

Interprety však mají i své výhody oproti kompilačním překladačům. Při výskytu chyby máme vždy přesné informace o jejím výskytu a můžeme poměrně rychle odhalit její příčinu. Tento přístup je tedy vhodný zvláště při ladění programů. Interprety umožňují modifikaci textu programu i během jeho činnosti, což se využívá často u jazyků jako je Prolog nebo LISP. U jazyků, které nemají blokovou strukturu (např. BASIC, APL), se může změnit některý příkaz, aniž by se musel znovu překládat zbytek programu. Interprety se dále používají tam, kde se mohou typy objektů dynamicky měnit v průběhu provádění programu — typickým příkladem je jazyk Smalltalk-80. Jejich zpracování je pro kompilační překladače značně

obtížné. Interpretační překladače bývají značně strojově nezávislé, neboť negenerují strojový kód. Pro přenos na jiný počítač obvykle postačí interpret znovu zkompilovat.

Uvedené dva přístupy jsou však extrémní, mnoho překladačů využívá spíše jejich kombinace. Některé interpretační překladače například nejdříve převedou zdrojový program do nějakého vnitřního tvaru (v nejjednodušším případě alespoň nahradí klíčová slova jejich binárními kódy) a ten potom interpretují. Výsledné řešení je kompromisem mezi časově náročným překladem kompilovaného a pomalým během interpretovaného programu.

Výběr vhodného přístupu, zda kompilovat nebo interpretovat, závisí obvykle na povaze jazyka a prostředí, ve kterém se používá. Pro časově náročné matematické výpočty se používají kompilační překladače, naopak pro účely výuky jazyků nebo na malých mikropočítačích se dává přednost interpretaci (typickými příklady takových jazyků jsou BASIC, LOGO nebo Smalltalk-80). Pro jazyk LISP se často používá zároveň obou přístupů, neboť jeho kompilace je časově značně náročná a kompilovaný program nemá dostatečné prostředky pro ošetření chyb. Interpretační překlad se také běžně užívá u různých příkazových jazyků, kdy se očekává okamžité provedení příkazu — příkladem mohou být dotazovací databázové jazyky jako SQL nebo jazyky řídicích programů, umožňující spouštění programů a komunikaci s operačním systémem počítače, např. `sh` nebo `csh` v systému Unix.

Tento učební text je orientován převážně na kompilační překladače, i když mnoho uvedených algoritmů je možné použít také při psaní interpretu. V obou případech bývá stejná analýza zdrojového kódu a často bývají podobné i metody hledání nejefektivnějšího kódu pro interpretaci s metodami generování cílového kódu.

### 1.1.3 Další použití překladačů

Techniky překladačů se samozřejmě nejčastěji používají pro překladače programovacích jazyků. Mají však mnohem širší využití i v jiných oblastech. Mnoho podpůrných programových prostředků, které manipulují se zdrojovým programem, provádí rovněž jistý druh analýzy. Tyto prostředky zahrnují například:

- *Strukturované editory.* Strukturovaný editor má jako vstup posloupnost příkazů pro vybudování zdrojového programu. Struktorovaný editor neprovádí pouze funkce pro vytváření a modifikaci textu jako běžný textový editor, ale analyzuje navíc text programu a vkládá do něj vhodnou hierarchickou strukturu. Strukturovaný editor tedy může plnit ještě další úkoly, které jsou užitečné při přípravě programu. Může například kontrolovat, zda je vstup správně syntakticky zapsán, může automaticky doplňovat klíčová slova (např. když uživatel napíše **while**, doplní editor odpovídající **do** a připomene uživateli, že mezi nimi musí být logický výraz) nebo může přecházet z klíčového slova **begin** nebo levé závorky na odpovídající **end** nebo pravou závorku. Navíc výstup takového editoru je často podobný výstupu analytické části překladače.

- *Formátovací programy.* Formátovací program (pretty printer) analyzuje program a tiskne ho takovým způsobem, aby byla zřetelná jeho struktura. Například poznámky mohou být vytištěny jiným typem písma a příkazy mohou být odsazeny v závislosti na úrovni jejich zanoření v hierarchické struktuře příkazů.

- *Programy pro sazbu textů.* Programy pro sazbu textů umožňují kombinovat text knihy, článku nebo dopisu s příkazy, které zajišťují členění na odstavce, kapitoly, změnu typu a velikosti písma, vytváření obsahu nebo indexu, speciální sazbu matematických textů nebo dokonce sazbu not nebo šachových partií. Typickým zástupcem této třídy programů jsou TEX a LATEX [8, 9], kterými byl připraven tento učební text.



Zdrojovým jazykem překladače nemusí být vždy nějaký programovací jazyk. Může se jednat také o některý přirozený jazyk (např. angličtinu), speciální jazyk popisující strukturu křemíkového integrovaného obvodu nebo strukturu grafických informací, které se mají zobrazit na tiskárně. Cílovým kódem takového překladače pak může být třeba jiný přirozený jazyk, maska integrovaného obvodu nebo posloupnost příkazů pro ovladač laserové tiskárny. Programovacím jazykem tohoto typu je například PostScript [1], který se používá pro vytváření grafiky, nebo Metafont [7], kterým se definují tvary znaků používaných při sazbě textů připravených programem TEX. Tyto jazyky mají i prostředky pro vytváření cyklů, podmíněných příkazů nebo pro definování vlastních procedur nebo funkcí.

V dalších kapitolách se budeme věnovat výhradně klasickým překladačům, opět s tím, že uvedené techniky jsou použitelné i v jiných oblastech, zejména techniky analýzy zdrojového textu.

## 1.2 Struktura překladače

Překladač musí provádět dvě základní činnosti: analyzovat zdrojový program a vytvářet k němu odpovídající cílový program. Analýza spočívá v rozkladu zdrojového programu na jeho základní součásti, na základě kterých se během syntézy vybudují moduly cílového programu. Obě části překladače, analytická i syntetická, využívají ke své činnosti společné tabulky.

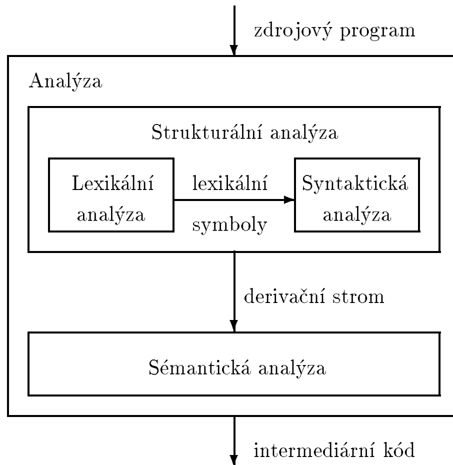
Analýza zdrojového programu při překladu probíhá na následujících třech úrovních:

- *Lexikální (lineární) analýza.* Zdrojový program vstupuje do procesu překladu jako posloupnost znaků. Tato posloupnost se čte lineárně zleva doprava a sestavují se z ní lexikální symboly (tokens) jako konstanty, identifikátory, klíčová slova nebo operátory.

- *Syntaktická (hierarchická) analýza.* Z posloupnosti lexikálních symbolů se vytvářejí hierarchicky zanořené struktury, které mají jako celek svůj vlastní význam, např. výrazy, příkazy, deklarace nebo program.

- *Sémantická analýza.* Během sémantické analýzy se provádějí některé kontroly, zajišťující správnost programu z hlediska vazeb, které nelze provádět v rámci syntaktické analýzy (např. kontrola deklarací, typová kontrola apod.).

Uvedené členění na úrovně analýzy vychází z toho, že běžné programovací jazyky jsou z hlediska Chomského klasifikace typu 1, tj. kontextové. Pro přímou analýzu kontextových jazyků dosud nebyly vyvinuty — na rozdíl od jazyků bezkontextových — dostatečně efektivní prostředky. Proto se na každé z těchto úrovní používají speciální metody specifikace i implementace, které využívají vlastností jazyků příslušných typů, tj. lineární analýza se provádí prostředky pro analýzu regulárních jazyků a hierarchická analýza prostředky pro analýzu bezkontextových jazyků. Pro sémantickou analýzu se obvykle využívá některá modifikace atributových gramatik, větší část sémantické analýzy však bývá implementována přímo prostředky jazyka, jímž je realizován překladač.




Obrázek 1.3: Struktura analytické části překladače

### 1.2.1 Lexikální analýza

Fáze lexikální analýzy (lexical analysis, scanning) čte znaky zdrojového programu a sestavuje je do posloupnosti *lexikálních symbolů*, v níž každý symbol představuje logicky související posloupnost znaků jako identifikátor nebo operátor obdobný :=. Posloupnost znaků tvořících symbol se nazývá *lexém* (lexeme).

Po lexikální analýze znaků např. v tomto přiřazovacím příkazu

`pozice := počátek + rychlost * 60` (1.1)

by se vytvořily následující lexikální jednotky:

1. identifikátor `pozice`
2. symbol přiřazení `:=`
3. identifikátor `počátek`
4. operátor `+`
5. identifikátor `rychlost`
6. operátor `*`
7. číslo `60`

Symboly, které zahrnují celou třídu lexikálních jednotek (identifikátor, číslo, řetězec), jsou reprezentovány obvykle jako dvojice <druh symbolu, hodnota>, přičemž druhá část dvojice může být pro některé symboly prázdná. Výstupem lexikálního analyzátoru pro příkaz (1.1) by tedy mohla být posloupnost

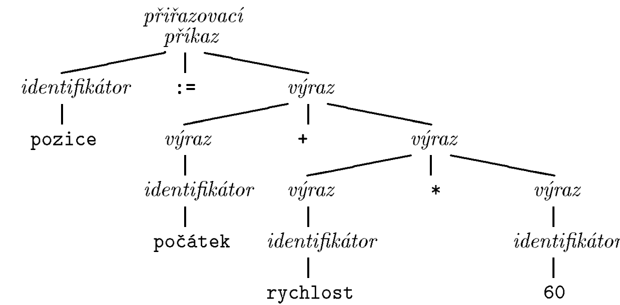


```
<id,pozice> <:=> <id,počátek> <+> <id,rychlost> <*> <num,60>
```

Mezery, konce řádků a poznámky oddělující lexikální symboly se obvykle během lexikální analýzy vypouštějí.

### 1.2.2 Syntaktický analyzátor

Syntaktická analýza (parsing, syntax analysis) spočívá v sestavování lexikálních jednotek ze zdrojového programu do gramatických frází, které překladač používá pro syntézu výstupu. Gramatické fráze zdrojového programu se obvykle reprezentují *derivačním stromem* obdobným stromu na obr. 1.4.




Obrázek 1.4: Derivační strom pro výraz `pozice:=počátek+rychlost*60`

Ve výrazu `počátek+rychlost*60` je fráze `rychlost*60` logickou jednotkou, neboť podle běžných matematických konvencí pro aritmetické výrazy se násobení provádí před sčítáním. Vzhledem k tomu, že za výrazem `počátek+rychlost` následuje `*`, nevytváří tento výraz v situaci na obr. 1.4 frázi.

Hierarchická struktura programu se obvykle vyjadřuje pomocí rekurzivních pravidel, zapsaných ve formě bezkontextové gramatiky. Například pro definici části výrazu můžeme mít následující pravidla:

```
výraz  ->  identifikátor        (1)
výraz  ->  číslo                (2)
výraz  ->  výraz + výraz        (3)
výraz  ->  výraz * výraz        (4)
výraz  ->  ( výraz )            (5)
```

Pravidla (1) a (2) jsou (nerekurzivní) základní pravidla, zatímco (3)–(5) definují výraz pomocí operátorů aplikovaných na jiné výrazy. Podle pravidla (1) jsou tedy `počátek` a `rychlost` výrazy. Podle pravidla (2) je `60` výraz, zatímco z pravidla (4) můžeme nejprve odvodit, že `rychlost*60` je výraz a konečně z pravidla (5) také `počátek+rychlost*60` je výraz.

Podobným způsobem jsou definovány příkazy jazyka, jako např.:

```
příkaz  ->  identifikátor := výraz
příkaz  ->  while ( výraz ) do příkaz          (1.2)
příkaz  ->  if ( výraz ) then příkaz
```

Dělení na lexikální a syntaktickou analýzu je dosti volné. Obvykle vybíráme takové rozdělení, které zjednodušuje činnost analýzy. Jedním z faktorů, které přitom uvažujeme, je to, zda jsou konstrukce zdrojového jazyka regulární nebo ne. Lexikální jednotky lze obvykle popsat jako regulární množiny, zatímco konstrukce vytvořené z lexikálních jednotek již vyžadují obecnější přístupy.

Typickou regulární konstrukcí jsou identifikátory, popsané obvykle jako posloupnosti písmen a číslic začínající písmenem. Běžně rozpoznáváme identifikátory jednoduchým prohlížením vstupního textu, v němž očekáváme znak, který není písmeno ani číslice, a potom seskupíme všechna písmena a číslice nalezené až do tohoto místa do lexikální jednotky pro identifikátor. Znaky takto shromážděné zaznamenáme do tabulky (tabulky symbolů) a odstraníme je ze vstupu tak, aby mohlo pokračovat zpracování dalšího symbolu.

Tento způsob lineárního prohledávání na druhé straně není dostatečný pro analýzu výrazů nebo příkazů. Nemůžeme například jednoduše kontrolovat dvojice závorek nebo klíčových slov `begin` a `end` v příkazech bez zavedení jakéhosi druhu hierarchické struktury na vstupu.

Derivační strom na obr. 1.4 popisuje syntaktickou strukturu vstupu, ale obsahuje informace, které nejsou důležité pro další průběh překladu. Mnohem běžnější vnitřní reprezentaci této syntaktické struktury dává *syntaktický strom* na obr. 1.5(a). Syntaktický strom je zhuštěnou reprezentací derivačního stromu; operátory v něm vystupují jako vnitřní uzly a operandy těchto operátorů jsou následníky jejich příslušných uzlů.

### 1.2.3 Sémantická analýza

Fáze sémantické analýzy zpracovává především informace, které jsou uvedeny v *deklaracích*, ukládá je do vnitřních datových struktur a na jejich základě provádí sémantickou kontrolu příkazů a výrazů v programu. K identifikaci operátorů a operandů těchto výrazů a příkazů využívá hierarchickou strukturu, určenou ve fázi syntaktické analýzy.

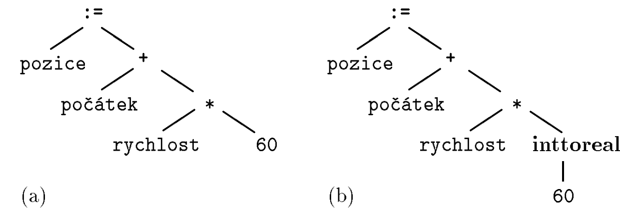
Důležitou složkou sémantické analýzy je *typová kontrola*. Kompilátor zde kontroluje, zda všechny operátory mají operandy povolené specifikací zdrojového jazyka. Mnoho definic programovacích jazyků například vyžaduje, aby kompilátor hlásil chybu, kdykoliv je reálné číslo použito jako index pole. Specifikace jazyka však může dovolit některé implicitní transformace operandů, například při aplikaci binárního aritmetického operátoru na celočíselný a reálný operand. V tomto případě může kompilátor požadovat konverzi celého čísla na reálné.

Jsou-li např. všechny proměnné v našem ukázkovém příkazu reálné, je třeba provést konverzi celočíselné konstanty `60` na reálnou, jak znázorňuje obr. 1.5(b). V tomto případě je rovněž možné typovou konverzi provést přímo a konstantu `60` nahradit hodnotou `60.0`.

### 1.2.4 Generování mezikódu

Po ukončení syntaktické a sémantické analýzy generují některé překladače explicitní *intermediární reprezentaci* zdrojového programu (*mezikód*). Intermediární reprezentaci můžeme považovat za program pro nějaký abstraktní počítač. Tato reprezentace by měla mít dvě důležité vlastnosti: měla by být jednoduchá pro vytváření a jednoduchá pro překlad do tvaru cílového programu.






Obrázek 1.5: Syntaktický strom

Intermediární kód slouží obvykle jako podklad pro optimalizaci a generování cílového kódu. Může však být také konečným produktem překladu v interpretačním překladači, který vygenerovaný mezikód přímo provádí.

Intermediární reprezentace mohou mít různé formy. Například tříadresový kód se podobá jazyku symbolických instrukcí pro počítač, jehož každé místo v paměti může sloužit jako registr. Tříadresový kód se skládá z posloupnosti instrukcí s nejvýše třemi operandy. Zdrojový program z (1.1) by mohl v tříadresovém kódu vypadat následovně:

```
temp1  := inttoreal(60)
temp2  := rychlost * temp1          (1.3)
temp3  := počátek + temp2
pozice := temp3
```

Intermediárními reprezentacemi, které se využívají v překladačích, se budeme zabývat v kapitole 8. Obecně tyto reprezentace musejí dělat více než jen výpočty výrazů; musejí si například poradit s řídicími konstrukcemi a voláním procedur.

### 1.2.5 Optimalizace kódu

Fáze optimalizace kódu se pokouší vylepšit intermediární kód tak, aby jeho výsledkem byl rychlejší nebo kratší strojový kód. Pojem "optimalizace" se nechápe doslovně jako nalezení nejlepší varianty, některé optimalizační algoritmy mohou ve zcela speciálních případech vést dokonce ke zhoršení vlastností původního kódu.

Některé optimalizace jsou triviální. Například přirozený algoritmus generuje intermediární kód (1.3) pomocí jedné instrukce pro každý operátor ve stromové reprezentaci po sémantické analýze, i když existuje lepší způsob provedení těchže výpočtů pomocí celkem dvou instrukcí:

```
temp1  := rychlost * 60.0
pozice := počátek + temp1          (1.4)
```

Uvedenou optimalizaci lze bez problémů v překladači realizovat. Překladač totiž může zjistit, že konverzi hodnoty `60` z celočíselného na reálný tvar lze provést jednou provždy v čase překladu, takže operaci `inttoreal` je možné vypustit. Dále hodnota `temp3` se používá pouze jednou pro přenesení její hodnoty do proměnné `pozice`. Můžeme tedy bez obav použít místo `temp3` přímo proměnnou `pozice`, takže není potřebný poslední příkaz v (1.3) a dostaneme kód (1.4).

V množství různých prováděných optimalizací se jednotlivé překladače od sebe značně liší. Překladače, tzv. "optimalizující," které provádějí většinu optimalizací, stráví podstatnou

část doby překladu právě v této fázi. Existují však i jednoduché optimalizace, které podstatně zlepší dobu běhu přeloženého programu bez velkého zpomalení překladu.

### 1.2.6 Generování cílového kódu

Poslední fází překladače je generování cílového kódu, což je obvykle přemístitelný strojový kód nebo program v jazyce asembleru. Všem proměnným použitým v programu se přidělí místo v paměti. Potom se instrukce mezikódu překládají do posloupnosti strojových instrukcí, které provádějí stejnou činnost. Kritickým problémem je přiřazení proměnných do registrů.

Překlad kódu (1.4) může například s použitím registrů 1 a 2 vypadat takto:

```
MOVF rychlost, R2
MULF #60.0, R2
MOVF počátek, R1
ADDF R2, R1
MOVF R1, pozice
```

První operand každé instrukci je zdrojový, druhý operand cílový. Písmeno `F` ve všech instrukcích znamená, že pracujeme s hodnotami v pohyblivé řádové čárce. Uvedený kód přesune obsah adresy `rychlost` (obešli jsme zatím důležitý problém přidělení paměti identifikátorům zdrojového programu) do registru 2, potom ho vynásobí reálnou konstantou `60.0`. Znak `#` znamená, že se má hodnota `60.0` zpracovat jako konstanta. Třetí instrukce přesouvá hodnotu `počátek` do registru 1 a přičítá k němu hodnotu vypočtenou dříve v registru 2. Na konec se přesune hodnota z registru 1 na adresu `pozice`, takže tento kód implementuje přiřazení z obr. 1.4.

### 1.2.7 Tabulka symbolů

Základní funkcí tabulky symbolů je zaznamenávání identifikátorů použitých ve zdrojovém programu a shromažďování informací a různých atributech každého identifikátoru. Tyto atributy mohou poskytovat informaci o paměti přidělené např. proměnné, její typu, rozsah platnosti a v případě jmen procedur takové věci jako počet a typy argumentů, způsob předávání každého argumentu (např. odkazem) a typ vrácené hodnoty, pokud nějaká existuje.

Tabulka symbolů (symbol table) je datová struktura obsahující pro každý identifikátor jeden záznam s jeho atributy. Tato datová struktura umožňuje rychlé vyhledání záznamu pro konkrétní identifikátor a rychlé ukládání nebo vybírání příslušných dat ze záznamu. Tabulkami symbolů se budeme zabývat v kapitole 5.

Rozpozná-li lexikální analyzátor ve zdrojovém programu identifikátor, může ho rovnou uložit do tabulky symbolů. Během lexikální analýzy však normálně nemůžeme všechny atributy identifikátoru určit. Například v pascalovské deklaraci

```
var pozice, počátek, rychlost : real;
```

není typ `real` znám v okamžiku, kdy lexikální analyzátor vidí identifikátory `pozice`, `počátek` a `rychlost`.

Informace o identifikátorech ukládají do tabulky symbolů zbývající fáze, které je také různým způsobem využívají. Během sémantické analýzy a generování intermediárního kódu například potřebujeme znát typy proměnných a funkcí, abychom mohli zkontrolovat jejich správné použití ve zdrojovém programu a generovat pro ně správné operace. Generátor kódu typicky ukládá a používá podrobné informace o paměti přidělené jednotlivým objektům v programu.



### 1.2.8 Diagnostika a protokol o průběhu překladu

Velkou část chyb zpracovávají fáze syntaktické a sémantické analýzy. Lexikální fáze odhaluje chyby v případě, že znaky na vstupu netvoří žádný symbol jazyka. Chyby, kdy posloupnost symbolů porušuje strukturní pravidla (syntaxi) jazyka, se detekují během syntaktické analýzy. Během sémantické analýzy se překladač pokouší nalézt konstrukce, které mají sice syntaktickou strukturu odpovídající bezkontextové gramatice jazyka, avšak porušují kontextová omezení (např. nedeklarované proměnné) nebo sémantická pravidla jazyka, např. pokud se pokoušíme sečíst v Pascalu dva identifikátory, z nichž jeden je jménem pole a druhý jménem procedury. Zpracováním chyb v jednotlivých fázích se budeme zabývat podrobněji vždy v příslušné kapitole.

Při výskytu chyby ve zdrojovém textu (případně chyby způsobené vnějšími okolnostmi, jako např. neúspěšný zápis do pracovního souboru v důsledku zaplnění disku) musí překladač nějakým způsobem reagovat. Možné reakce překladače můžeme obecně shrnout do následujícího seznamu:

I. Nepřijatelné reakce
   (a) Nesprávné reakce (bez ohlášení chyby)
      - Překladač *zhavaruje* nebo *cyklí*.
      - Překladač *pokračuje*, ale generuje nesprávný cílový program.
   (b) Správné reakce (ale nepoužitelné)
      - Překladač nahlásí první chybu a *zastaví se*.

II. Přijatelné reakce
   (a) Možné reakce
      - Překladač nahlásí chybu a *zotaví se*, pokračuje v hledání dalších možných chyb.
      - Překladač nahlásí a *odstraní chybu*, pokračuje v generování správného cílového kódu.
   (b) Nemožné reakce (se současnými metodami)
      - Překladač nahlásí a *opraví chybu*, pokračuje v generování programu odpovídajícího přesně záměrům programátora.

Nejproblematičtější je případ, kdy překladač na chybu nezareaguje a vytvoří cílový kód. Taková chyba se může projevit až po delší době a může způsobit i vážnou ztrátu dat. Přeložený program může mít neočekávané chování, které není vysvětlitelné na základě jeho zdrojového kódu. Ukončení překladu po první nalezené chybě značně prodlužuje proces ladění programu nutností neustále opakovaných překladů. Tento typ reakce je snad ještě možný v integrovaných vývojových prostředích, kdy se oprava a nové spuštění programu provede velmi jednoduše, ale obecně lze říci, že minimální přijatelnou reakcí překladače na chybu je zotavení. Algoritmy, které umožňují odstranění chyb (modifikací zdrojového textu nebo vnitřního tvaru programu) jsou časově náročné a tedy nevhodné pro interaktivní prostředí. Navíc umožňují spuštění nesprávně modifikovaného programu s možnými důsledky jako při neohlášení chyby.

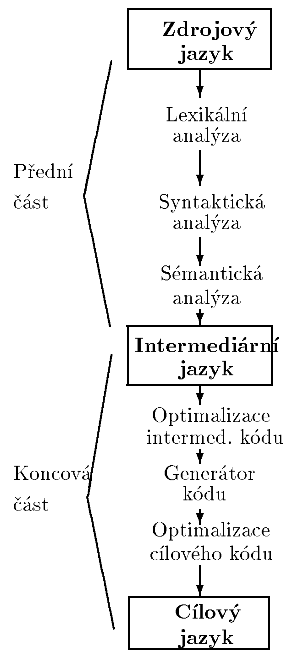
## 1.3 Organizace překladu

### 1.3.1 Fáze překladu

Obecné schéma překladače z hlediska jeho členění na fáze je uvedeno na obr. 1.6. Toto členění odpovídá logické struktuře překladače, která však nemusí přímo odpovídat skutečné implementaci.

Jednotlivé fáze se často rozdělují na *přední část* (front end) a *koncovou část* (back end). Přední část se skládá z těch fází nebo jejich částí, které závisejí převážně na zdrojovém jazyku a jsou dosti nezávislé na cílovém počítači. Obvykle zahrnuje lexikální a syntaktickou analýzu, vytváření tabulky symbolů, sémantickou analýzu a generování intermediárního kódu. V přední části překladače lze provést rovněž jistou část optimalizace kódu. Obsahuje také obsluhu chyb, které vznikají během analýzy.

Koncová část zahrnuje ty části překladače, které již závisejí na cílovém počítači, a obecně nezávisí na zdrojovém jazyku, ale na intermediárním kódu. V koncové části překladače nalezneme prvky fáze optimalizace kódu a generování kódu společně s nutnými operacemi pro obsluhu chyb a operace s tabulkou symbolů. Dělení na přední a koncovou část obvykle koresponduje s dělením na analytickou a syntetickou část překladu, i když přední část také provádí syntézu intermediárního kódu a koncová část zase tento kód analyzuje.




Obrázek 1.6: Fáze překladače



Při přenosu překladače na jiný cílový počítač se při dobře provedeném návrhu pouze převezme přední část, ke které se připojí nově vytvořená koncová část. Je-li koncová část vhodně navržena, nemusí dokonce být nutné ji příliš měnit. V rozsáhlejších návrhových systémech s více jazyky se někdy také snažíme překládat několik různých programovacích jazyků do téhož intermediárního jazyka a použít pro různé přední části jedinou koncovou část. Vzhledem k tomu, že ale mezi koncepcemi různých jazyků existují určité rozdíly, má tento postup jen omezené možnosti. Uvedený postup zvolila např. firma JPI ve své řadě překladačů TopSpeed (C, C++, Pascal, Modula-2).

### 1.3.2 Průchody

Několik fází překladu se obvykle implementuje do jediného *průchodu* (pass) skládajícího se ze čtení vstupního souboru a zápisu výstupního souboru. V praxi existuje mnoho variací ve způsobu rozdělení fází překladače do průchodů, které závisejí především na následujících okolnostech:

- Vlastnosti zdrojového a cílového jazyka.
- Velikost dostupné paměti pro překlad.
- Rychlost a velikost překladače.
- Rychlost a velikost cílového programu.
- Požadované informace a prostředky pro ladění.
- Požadované techniky detekce chyb a zotavení.
- Rozsah projektu — velikost programátorského týmu, časové možnosti.

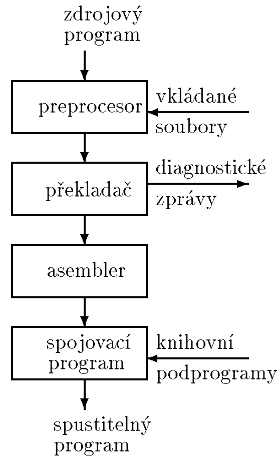
Překladače určené především pro výuku jsou obvykle jednoprůchodové. Neprovádějí mnoho optimalizací, neboť se předpokládá častější spouštění překladače než samotného přeloženého programu. Větší důraz se u nich klade na zpracování chyb a možnosti ladění. Naopak v překladačích používaných pro vytváření uživatelských aplikací je důležitá důkladná optimalizace, která se obvykle provádí ve více průchodech. Některé jazyky dokonce není možné překládat v jednom průchodu z prinicipiálních důvodů, neboť například umožňují volat procedury dříve, než jsou známy typy jejich parametrů.

Činnost fází, které vytvářejí jeden průchod, se často navzájem překrývá. Například lexikální, syntaktická a sémantická analýza mohou vytvářet jediný průchod. Posloupnost symbolů po lexikální analýze pak můžeme překládat přímo do intermediárního kódu. Syntaktický analyzátor můžeme při podrobnějším pohledu brát jako řídicí prvek. Pokouší se odkrýt gramatickou strukturu symbolů, které vidí; symboly získává tehdy, když je potřebuje, voláním lexikálního analyzátoru. Po rozpoznání gramatické struktury syntaktický analyzátor volá generátor intermediárního kódu, aby provedl sémantickou analýzu a vygeneroval část kódu. Náš pohled na návrh překladače bude směřovat právě k tomuto způsobu organizace.

## 1.4 Příbuzné programy

K překladači mohou být navíc nutné pro vytvoření proveditelného programu i některé další pomocné programy (viz obr. 1.7). Typický proces zpracování zdrojového programu v sobě

může zahrnovat spuštění preprocesoru, který zpracuje makrodefinice, příkazy pro podmíněný překlad nebo příkazy pro vložení textu z jiného souboru do zdrojového programu. Po překladu vznikne cílový kód, který může mít buď tvar přemístitelného binárního modulu, nebo v některých jednodušších překladačích může být výstupem program, který je třeba dále zpracovat asemblerem. Přeložené moduly musí dále zpracovat spojovací program, který k nim připojí knihovní podprogramy a obvykle i část kódu, která zajišťuje různé pomocné činnosti v době běhu programu (tzv. run-time systém). Výsledkem činnosti spojovacího programu je již spustitelný program.




Obrázek 1.7: Postup při vytváření spustitelného programu

Některé rozsáhlejší vývojové systémy obsahují kromě uvedených základních prostředků ještě různé podpůrné programy, zajišťující například tyto činnosti:

- Ladění programu na symbolické nebo strojové úrovni.
- Zkoumání uschovaného obsahu paměti po havárii programu.
- Zpětný překlad cílového programu do zdrojového tvaru.
- Formátování programu pro tisk.
- Tisk seznamu křížových referencí.
- Generování statistik o činnosti programu (profilování) — např. počet volání každé procedury, využití operační paměti, času procesoru apod.
- Archivace vývojových verzí programu.
- Údržba aktuální verze programu — automatické spouštění překladu změněných programových modulů a budování spustitelného programu (programy typu `make`).



- Údržba knihoven podprogramů.
- Specializované editory.

Při návrhu překladače je třeba mít použití těchto prostředků na paměti tak, aby jich mohl uživatel co nejvíce využívat. Překladač například musí zajistit generování dostatečných informací pro symbolické ladění programu (jména a umístění proměnných a procedur, odkazy na začátky zdrojových řádků apod.) nebo musí do generovaného programu vkládat volání speciálních služeb pro vyhodnocování statistik o činnosti programu.

## 1.5 Automatizace výstavby překladačů

V rámci teorie a praktických aplikací byla vyvinuta řada programových nástrojů, které usnadňují implementaci překladačů. Jejich spektrum zahrnuje jednoduché generátory (konstruktory) lexikálních a syntaktických analyzátorů, ale i komplexní systémy nazývané *generátory překladačů* (compiler-generators), *kompilátory kompilátorů* (compiler-compilers) nebo *systémy pro psaná překladačů* (translator-writing systems). Tyto systémy na základě specifikace zdrojového jazyka a cílového počítače generují překladač pro daný jazyk. Vstupní specifikace může zahrnovat

- popis lexikální a syntaktické struktury zdrojového jazyka,
- popis, co se má generovat pro každou konstrukci zdrojového jazyka,
- popis počítače, pro který má být generován kód.

V mnoha případech jsou tyto specifikace v podstatě souborem programů, které generátor kompilátorů vhodně "spojí." Některé generátory však umožňují, aby části specifikací měly neprocedurální charakter, tj. aby například namísto sytaktického analyzátoru mohl tvůrce zadat pouze bezkontextovou gramatiku a generátor sám převede tuto gramatiku na program realizující syntaktickou analýzu zdrojového jazyka. Všechny tyto systémy však mají určitá omezení. Problém spočívá v kompromisu mezi množstvím práce, které dělá generátor kompilátoru automaticky, a pružností celého systému. Ilustrujme tento problém na příkladě lexikálního analyzátoru.

Většina systémů pro psaní překladačů dodává ve skutečnosti tentýž podprogram lexikální analýzy pro generovaný kompilátor, lišící se pouze v seznamu klíčových slov specifikovaných uživatelem. Pro většinu případů je toto řešení vyhovující, problém však nastane v případě nestandardní lexikální jednotky, např. identifikátoru, který může kromě číslic a písmen obsahovat i jiné znaky. I když existuje obecnější přístup k automatické konstrukci tohoto analyzátoru (reprezentovaný například generátorem `lex`, kterému se budeme věnovat podrobněji v článku 2.6), větší pružnost systému vyžaduje podrobnější specifikaci a tudíž i více práce.

K základním možnostem existujících generátorů překladačů patří:

- generátor lexikálního analyzátoru,
- generátor syntaktického analyzátoru a
- prostředky pro generování kódu.

Principy činnosti a výstavby obou generátorů analyzátorů jsou založeny na teorii formálních jazyků a gramatik. Podstatnou výhodou použití těchto generátorů je zvýšení spolehlivosti překladače. Mechanicky generované části překladače jsou daleko méně zdrojem chyb než části programované ručně.

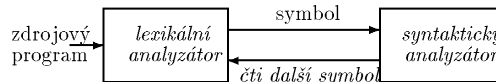
Jako prostředků usnadňujících generování kódu se v těchto systémech obvykle používá vyššího programovacího jazyka. Slouží ke specifikaci generování jak intermediárního kódu, tak i symbolických instrukcí nebo strojového jazyka. Ve tvaru např. sémantických podprogramů jsou pak tyto specifikace volány automaticky generovaným syntaktickým analyzátorem na vhodných místech. Mnoho systémů pro psaní překladačů používá také mechanismu pro zpracování rozhodovacích tabulek, které vybírají generovaný cílový kód. Tyto tabulky jsou spolu s jejich interpretem generovány na základě popisu vlastností cílového jazyka a tvoří součást výsledného kompilátoru.

# Kapitola 2

# Lexikální analýza

## 2.1 Činnost lexikálního analyzátoru

Lexikální analyzátor je první fází překladače. Jeho hlavním úkolem je číst znaky ze vstupu a na svůj výstup dávat symboly, které dále používá syntaktický analyzátor. Tato interakce, schematicky shrnutá na obr. 2.1, se běžně implementuje tak, že lexikální analyzátor vytvoříme jako podprogram nebo koprogram syntaktického analyzátoru. Po přijetí příkazu "dej další symbol" od syntaktického analyzátoru čte lexikální analyzátor vstupní znaky až do té doby, než může identifikovat další symbol.




Obrázek 2.1: Interakce lexikálního a syntaktického analyzátoru

Vzhledem k tomu, že lexikální analyzátor je tou částí překladače, která čte zdrojový text, může na uživatelském rozhraní provádět i další úkoly. Jedním takovým úkolem je odstraňování poznámek a odsazovačů (mezer, tabelátorů a konců řádků) ze zdrojového programu. Dalším úkolem je udržování konzistence chybových hlášení překladače a zdrojového textu. Lexikální analyzátor může například sledovat počet načtených znaků konce řádku a umožnit ke každému chybovému hlášení připojení čísla příslušného řádku s chybou. V některých překladačích je lexikální analyzátor pověřen prováděním opisu zdrojového programu s vyznačenými chybovými hlášeními. Pokud zdrojový jazyk obsahuje některé funkce makroprocesoru, potom tyto funkce mohou být implementovány během lexikální analýzy.

Pro rozdělení analytické fáze překladu na lexikální analýzu a syntaktickou analýzu existuje několik důvodů.

1. Zřejmě nejpodstatnějším důvodem je jednodušší návrh překladače. Oddělení lexikální a syntaktické analýzy často umožňuje jednu nebo obě fáze zjednodušit. Například syntaktický analyzátor zahrnující i konvence pro poznámky a mezery je podstatně složitější než analyzátor, který předpokládá, že poznámky a mezery už byly odstraněny lexikálním analyzátorem.





| SYMBOL | PŘÍKLADY LEXÉMŮ | NEFORMÁLNÍ POPIS VZORU |
|---|---|---|
| **const** | `const` | `const` |
| **if** | `if` | `if` |
| **relation** | `<, <=, =, <>, >, >=` | `<` nebo `<=` nebo `=` nebo `<>` nebo `>=` nebo `>` |
| **id** | `pi, count, D2` | písmeno následované písmeny a číslicemi |
| **num** | `3.1416, 0, 6.02E23` | libovolná číselná konstanta |
| **literal** | `"core dumped"` | libovolné znaky v uvozovkách kromě uvozovek |

Obrázek 2.2: Příklady symbolů

2. Zlepší se efektivita překladače. Oddělený lexikální analyzátor umožňuje použít specializované a potenciálně mnohem efektivnější algoritmy. Čtením zdrojového programu a jeho rozdělováním do symbolů se ztrácí mnoho času. Specializované techniky práce s vyrovnávací pamětí při čtení vstupních znaků mohou podstatně zvýšit výkonnost překladače.

3. Zvýší se přenositelnost překladače. Zvláštnosti vstupní abecedy a jiné anomálie konkrétních vstupních zařízení se mohou omezovat pouze na lexikální analyzátor. Například jazyk C umožňuje použití speciálních tříznakových kombinací pro znaky, které nebývají dostupné na některých klávesnicích ('??(' pro '[', '??<' pro '{' apod.).

Pro podporu automatizace vytváření oddělených lexikálních a syntaktických analyzátorů byly vytvořeny specializované prostředky. S programem Lex se seznámíme v této kapitole, programu Yacc bude věnována část kapitoly následující.

## 2.2 Základní pojmy

### 2.2.1 Symboly, vzory, lexémy

Když hovoříme o lexikální analýze, používáme výrazů *symbol*, *vzor* a *lexém* se specifickým významem. Příklady jejich použití ukazuje obrázek 2.2. Obecně existuje množina vstupních řetězců, pro které se na výstup dává týž symbol. Tato množina je popsána pravidlem zvaným vzor symbolu. Lexém je posloupnost znaků zdrojového programu, která odpovídá vzoru pro konkrétní symbol. Například v příkazu jazyka Pascal

```
const pi = 3.1416;
```

je podřetězec *pi* lexémem pro symbol *identifikátor*.

Symboly považujeme za terminální symboly gramatiky zdrojového jazyka. Lexémy odpovídající vzorům pro symboly představují řetězce znaků zdrojového programu, které můžeme považovat za jedinou lexikální jednotku.

V mnoha programovacích jazycích se za symboly považují následující konstrukce: klíčová slova, operátory, identifikátory, konstanty, řetězce (ve smyslu literálů) a interpunkční symboly jako závorky, čárky a středníky. Ve výše uvedeném příkladu se při výskytu posloupnosti znaků `pi` ve zdrojovém programu vrátí syntaktickému analyzátoru symbol reprezentující identifikátor. Vracení symbolů se často implementuje jako vracení celých čísel, která jsou symbolům přidělena (případně hodnot výčtového typu, pokud to implementační jazyk umožňuje).

Vzor je pravidlo popisující množinu lexémů, které mohou představovat ve zdrojovém programu konkrétní symbol. Vzor pro symbol **const** na obr. 2.2 je právě jediný řetězec `const`,

jímž je klíčové slovo označeno. Vzor pro symbol **relation** je množina relačních operátorů jazyka Pascal. Pro přesný popis mnohem složitějších symbolů jako je **id** (pro identifikátor) a **num** (pro číslo) budeme používat regulárních výrazů.

Některé jazykové konvence mají dopad na složitost lexikální analýzy. Jazyky jako Fortran vyžadují, aby určité konstrukce byly na pevné pozici ve vstupním řádku. Umístění lexému může být tedy důležité při určování správnosti zdrojového programu. Trend tvorby moderních programovacích jazyků směřuje ke vstupu ve volném formátu, který umožňuje umístění konstrukcí kdekoliv na vstupním řádku, takže tento aspekt lexikální analýzy se stává stále méně důležitým.

Zpracování mezer se značně jazyk od jazyka liší. V některých jazycích jako je Fortran, Basic nebo Algol 68 nejsou mezery v příkazech programu významné, až na mezery uvnitř literálových řetězců. Mohou být doplněny pro zvýšení čitelnosti programu. Konvence týkající se mezer mohou značně komplikovat úkol identifikace symbolů.

Populárním příkladem, který dokumentuje potenciální obtíže při rozpoznávání symbolů, je příkaz `DO` ve Fortranu. V příkazu

```
DO 5 I = 1.25
```

až do okamžiku, než uvidíme desetinnou tečku, nemůžeme poznat, že `DO` není klíčové slovo, ale část identifikátoru `DO5I`. Na druhé straně v příkazu

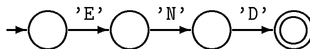
```
DO 5 I = 1,25
```

máme sedm symbolů, které odpovídají klíčovému slovu `DO`, návěští příkazu `5`, identifikátoru `I`, operátoru `=`, konstantě `1`, čárce a konstantě `25`. Zde si nemůžeme být až do výskytu čárky jistí, zda je `DO` klíčové slovo.

V mnoha jazycích jsou některé řetězce rezervovány, tj. jejich význam je předdefinován a nemůže být uživatelem změněn. Nejsou-li klíčová slova rezervována, musí klíčové slovo od uživatelem definovaného identifikátoru rozlišit lexikální analyzátor. V jazyce PL/I nejsou klíčová slova rezervovaná; pravidla pro rozlišení klíčových slov od identifikátorů jsou tedy značně komplikovaná, jak ukazuje následující příkaz PL/I:

```
IF THEN THEN THEN =  ELSE; ELSE ELSE = THEN;
```

Pro analýzu klíčových slov můžeme použít v podstatě dvou přístupů. Můžeme je definovat jako samostatné symboly se svou vlastní strukturou, např. klíčové slovo `END` jako řetězec




nebo můžeme pro klíčová slova použít stejného vzoru jako pro identifikátory a teprve po rozpoznání identifikátoru otestovat na základě tabulky klíčových slov, zda se jedná skutečně o identifikátor nebo o klíčové slovo a podle toho vrátit příslušný kód symbolu. Druhý přístup je výhodnější z hlediska složitosti automatu a pro většinu moderních jazyků zřejmě nemá smysl používat přístup první.

### 2.2.2 Atributy symbolů

Odpovídá-li vzoru více jak jeden lexém, musí lexikální analyzátor následujícím fázím překladače poskytnout informaci o tom, který konkrétní lexém byl rozpoznán. Například řetězcům `0` a `1` odpovídá vzor pro **num**, avšak pro generátor kódu je podstatné znát, o který řetězec se skutečně jedná.



Lexikální analyzátor shromažďuje informace o symbolech v atributech symbolů. Symboly mají vliv na rozhodování syntaktického analyzátoru; atributy ovlivňují překlad symbolů. V praxi má symbol často pouze jeden atribut — ukazatel na položku tabulky symbolů, která obsahuje informace o symbolu. Pro účely diagnostiky nás může zajímat jak lexém identifikátoru, tak i číslo řádku, na kterém se poprvé objevil. Obě tyto informace mohou být rovněž uloženy v položce tabulky symbolů pro identifikátor.

**Příklad 2.1.** Symboly a k nim příslušné hodnoty atributů pro příkaz jazyka Fortran

```
E = M * C ** 2
```

jsou uvedeny dále jako posloupnost dvojic:

```
<id, ukazatel na položku tabulky symbolů pro E>
<assign_op,>
<id, ukazatel na položku tabulky symbolů pro M>
<mult_op,>
<id, ukazatel na položku tabulky symbolů pro C>
<exp_op,>
<num, celočíselná hodnota 2>
```

Povšimněte si, že některé dvojice nemusejí obsahovat hodnotu atributu; první složka je dostatečná pro identifikaci lexému. V tomto malém příkladu dostal symbol **num** atribut s celočíselnou hodnotou. Překladač také může uložit řetězec znaků, který tvoří číslo, do tabulky symbolů a jako atribut symbolu **num** ponechat ukazatel na položku tabulky. ■

## 2.3 Vstup zdrojového textu

Na začátku této kapitoly jsme uvedli, že jedním z úkolů lexikálního analyzátoru je čtení znaků ze vstupního (zdrojového) souboru. Čtení znaků můžeme realizovat v nejjednodušším případě např. voláním standardní funkce `getchar()` jazyka C nebo procedury `read()` jazyka Pascal. Obecně se však jedná o podstatně složitější problém, a to z následujících důvodů:

- čtení po jednotlivých znacích může být značně neefektivní ve srovnání se čtením po řádcích nebo po velkých blocích textu, např. může představovat volání funkce jádra operačního systému se všemi kontrolami, které k tomu příslušejí. Analyzátor tedy musí zajistit nějakou správu vyrovnávacích pamětí, ze kterých se budou dále jednotlivé znaky odebírat. Ani prostředky vyrovnávaného vstupu dat, které poskytují standardní knihovny jazyka C, nejsou z hlediska efektivity dostatečné; v mnoha implementacích se čtená data kopírují až třikrát před tím, než je obdrží uživatelský program (z disku do vyrovnávací paměti operačního systému, dále do vyrovnávací paměti, která je částí struktury `FILE`, a nakonec do řetězce, který obsahuje lexém).

- při čtení zdrojového textu se může provádět jeho opis do výstupní tiskové sestavy, přičemž tento opis se může dále doplňovat o informace získané při překladu (úroveň zanoření závorkových struktur, adresy instrukcí apod.). I tehdy, když se opis celého zdrojového textu neprovádí, musí lexikální analyzátor udržovat pro účely hlášení chyb alespoň informaci o čísle zdrojového řádku, případně text aktuálního řádku a současnou pozici).

- v případě, že jazyk umožňuje vkládání částí zdrojového textu z jiných souborů, podmíněný překlad nebo práci s makrodefinicemi (např. jazyk C), je třeba tuto činnost, která může být značně složitá, provést buď jako samostatný průchod před lexikální analýzou, nebo se musí provést zároveň s činností lexikálního analyzátoru, a to právě během čtení znaků.
- během analýzy často potřebujeme provést návrat ve vstupním souboru; v případě, že nám implementační jazyk návrat neumožňuje nebo jsou-li možnosti navracení omezené (např. funkce `ungetc()` jazyka C umožňuje vrátit pouze jediný znak), je třeba tuto akci provádět ve vlastní režii.

Čtení zdrojového textu je vhodné implementovat jako samostatný programový modul komunikující s lexikálním analyzátorem přes určité rozhraní. Oddělením činností spojených se čtením zdrojového textu můžeme dosáhnout větší přenositelnosti překladače, neboť většina systémově závislých operací se soustřeďuje právě do vstupního modulu.

**Příklad 2.2.** Následující program je velmi jednoduchým příkladem implementace vstupního modulu. Definuje funkci `getch()`, která poskytuje následující znak ve vstupním souboru, a funkci `ungetch()` pro návrat o znak zpět. Dále jsou k dispozici proměnné obsahující číslo současného zdrojového řádku, text tohoto řádku a ukazatel na znak, který bude zpracován jako následující. Tyto informace lze dále využít pro hlášení chyb.

```
int  line = 0;            /* číslo zdrojového řádku   */
char source[ 256 ];       /* zdrojový řádek           */
char *gchptr = source;    /* ukazatel současné pozice */

int getch( void )     /* čtení jednoho znaku */
{
   char ch;
   if( *gchptr == '\0' ) {    /* jsme za koncem řádku     */
      gchptr = gets( source );
      if( gchptr == NULL )     /* konec zdrojového souboru */
         return( EOF );
      line++;
   }
   return (ch = *gchptr++) ? ch : '\n';
}

void ungetch( void )    /* návrat o znak zpět */
{
   if( gchptr != source ) {  /* nejsme na začátku řádku  */
      gchptr--;
}
```

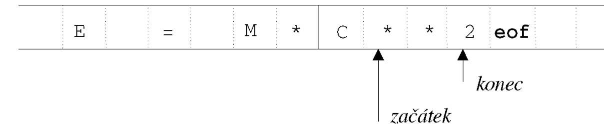
■

V některých programovacích jazycích je často třeba, aby měl lexikální analyzátor možnost si prohlédnout několik znaků za lexémem ještě před tím, než může spolehlivě ohlásit, o který symbol se jedná. Podprogramy `getch` a `ungetch` z příkladu 2.2 například umožňovaly přečíst znaky nejvýše do konce řádku a pak je zase vrátit zpět. Vzhledem k tomu, že neustálým



přesouváním znaků může docházet ke značným časovým ztrátám, používají se specializované techniky pracující s vyrovnávacími pamětmi (v příkladu 2.2 jsme měli vyrovnávací paměť na jeden zdrojový řádek). Tyto techniky jsou obvykle značně závislé na vlastnostech konkrétního operačního systému, proto pouze naznačíme jednu z možností.

Pro vstup zdrojového textu můžeme využít vyrovnávací paměti rozdělené na dvě části o velikosti $N$ znaků (viz obr. 2.3). Typická hodnota $N$ je daná velikostí diskového bloku, např. 1024 nebo 4096 slabik. Do každé poloviny načteme $N$ znaků textu, a to vždy jedním voláním operace čtení pro celý blok, ne pro jednotlivé znaky. Zbývá-li na vstupu méně než $N$ znaků, uloží se do vyrovnávací paměti za poslední načtený znak speciální znak **eof**.




Obrázek 2.3: Rozdělená vstupní vyrovnávací paměť

Pro přístup do vyrovnávací paměti budeme udržovat dva ukazatele Na počátku budou oba ukazatele ukazovat na tentýž znak; během analýzy bude jeden ukazatel označovat pozici prvního znaku lexému a druhý se bude přesunovat tak dlouho, až se nalezne konec lexému. Řetězec znaků mezi oběma ukazateli potom představuje současný lexém; po jeho zpracování se oba ukazatelé přesunou za konec lexému a činnost se opakuje.

Jestliže se ukazatel konce lexému má přesunout do pravé poloviny vyrovnávací paměti, naplní se pravá polovina dalšími $N$ znaky. Má-li se ukazatel přesunout za pravý konec vyrovnávací paměti, naplní se levá polovina dalšími $N$ znaky a ukazatel se přesune cyklicky na začátek vyrovnávací paměti.

Toto schéma umožňuje jen omezenou délku pohledu vpřed ve vstupním textu — omezení je dáno velikostí vyrovnávací paměti. Pokud však délka prohledávaného řetězce nepřekročí velikost vyrovnávací paměti, je vždy zajištěno, že se můžeme vrátit na začátek lexému. To je výhodné například tehdy, jestliže pro rozpoznání určité konstrukce potřebujeme znát širší kontext, v němž je tato konstrukce uvedena. V praxi se mohou používat některé další modifikace, které dále zvyšují efektivitu čtení zdrojového textu.

## 2.4 Specifikace a rozpoznávání symbolů

Při implementaci lexikálního analyzátoru vždy vycházíme z více či méně formálního popisu struktury jednotlivých lexikálních jednotek. Tento popis může být v jednom z následujících tvarů:

1. slovní popis,
2. regulární nebo lineární gramatika,
3. graf přechodů konečného automatu,
4. regulární výraz, resp. regulární definice.

Všechny tyto možnosti se v praxi vyskytují a až na případně možnou nejednoznačnost slovního popisu jsou rovnocenné. V dalších dvou odstavcích se budeme zabývat posledními dvěma variantami. Regulární nebo obecně lineární gramatiky lze snadno převést na konečný automat, podobně jako slovní popis struktury jazyka.

### 2.4.1 Regulární výrazy

Regulární výrazy jsou důležitou notací pro specifikaci vzorů symbolů. Každý vzor odpovídá množině řetězců, takže regulární výraz slouží vlastně jako pojmenování množiny řetězců. Článek 2.6 tuto notaci rozšiřuje na jazyk pro specifikaci lexikálních analyzátorů.

Regulární množiny byly formálně definovány v [12]. Pro naše účely si definici rozšíříme o některé velmi často se vyskytující konstrukce. Regulární výrazy nad abecedou $\Sigma$ a jazyky jimi označované budeme definovat následujícím způsobem:

1. $\epsilon$ je regulární výraz označující $\{\epsilon\}$, tj. množinu obsahující prázdný řetězec.
2. Je-li $a$ symbol v $\Sigma$, potom `a` je regulární výraz označující $\{a\}$, tj. množinu obsahující řetězec $a$. Ačkoliv pro tři různé významy používáme stejný zápis, je ve skutečnosti regulární výraz `a` odlišný od řetězce $a$ nebo od symbolu $a$. Z kontextu bude vždy zřejmé, zda hovoříme o regulárním výrazu, řetězci nebo symbolu.
3. Jsou-li $a, b, c, \ldots$ symboly v $\Sigma$, potom `[abc...]` je regulární výraz označující jazyk $\{a, b, c, \ldots\}$. Tvoří-li symboly posloupnost, lze je zapsat jako interval, např. `[a-z]`.
4. Předpokládejme, že $r$ a $s$ jsou regulární výrazy, které označují jazyky $L(r)$ a $L(s)$. Potom
   a) `(`$r$`)|(`$s$`)` je regulární výraz označující $L(r) \cup L(s)$,
   b) `(`$r$`)(`$s$`)` je regulární výraz označující $L(r)L(s)$,
   c) `(`$r$`)*` je regulární výraz označující $(L(r))^*$,
   d) `(`$r$`)+` je regulární výraz označující $(L(r))^+$,
   e) `(`$r$`)?` je regulární výraz označující $L(r) \cup \{\epsilon\}$,
   f) `(`$r$`)` je regulární výraz označující $L(r)$. (Toto pravidlo říká, že kolem regulárního výrazu můžeme podle potřeby napsat dvojici závorek.)

**Příklad 2.3.** Jazyk tvořený řetězci nul a jedniček s lichou paritou (tj. s lichým počtem jedniček) můžeme popsat regulárním výrazem

```
0*1(0*10*1)*0*
```

■

Regulární výrazy mohou popisovat pouze relativně jednoduché konstrukce. Některé jazyky nelze regulárními výrazy popsat, například v následujících situacích:

- Regulární výrazy nelze použít k popisu vyvážených nebo vnořených konstrukcí. Například množina všech řetězců s vyváženými závorkami se nedá regulárním výrazem popsat, stejně jako zanořené poznámky v jazyce Modula-2. Na druhé straně lze takové množiny popsat bezkontextovou gramatikou.



- Regulárními výrazy nelze popsat opakované řetězce. Množina

  $$\{wcw | w \text{ je řetězec symbolů } a \text{ a } b\}$$

  se nedá popsat regulárním výrazem ani bezkontextovou gramatikou.
- Regulární výrazy lze použít pouze k popisu pevného počtu opakování nebo nespecifikovaného počtu opakování dané konstrukce. Nelze porovnat dvě libovolná čísla, zda jsou stejná. Nemůžeme tedy pomocí regulárních výrazů popsat hollerithovské řetězce tvaru nHa1a2…an z prvních verzí jazyka Fortran, neboť počet znaků následujících za H musí odpovídat desítkovému číslu před H.

Vzhledem k tomu, že většina lexikálních konstrukcí běžných programovacích jazyků patří do třídy regulárních jazyků, je použití regulárních výrazů typické právě pro tuto oblast, neboť jejich analýza je podstatně jednodušší než analýza jazyků bezkontextových nebo kontextových.

### 2.4.2 Regulární definice

Pro účely zápisu bychom chtěli regulární výrazy pojmenovat a jejich jména použít v jiných regulárních výrazech, jako by to byly symboly. Je-li $\Sigma$ abeceda základních symbolů, potom regulární definice je posloupnost definic ve tvaru

$$\begin{aligned} d_1 &\longrightarrow r_1 \\ d_2 &\longrightarrow r_2 \\ &\cdots \\ d_n &\longrightarrow r_n \end{aligned}$$

kde $d_i$ jsou navzájem odlišná jména a $r_i$ jsou regulární výrazy nad abecedou $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$, tj. z množiny základních symbolů a dříve definovaných jmen. Omezením $r_i$ pouze na symboly množiny $\Sigma$ a dříve definovaná jména můžeme vytvořit regulární výraz nad $\Sigma$ pro každé $r_i$ opakovaným nahrazováním jmen regulárních výrazů výrazy, které označují. Je-li $r_i$ použito v $d_j$ pro nějaké $j \geq i$, potom by mohlo být $r_i$ definováno rekurzivně a tento proces nahrazování by se nezastavil.

Pro odlišení jmen od symbolů budeme psát jména v regulárních definicích kurzívou.

**Příklad 2.4.** Identifikátory jazyka Pascal můžeme popsat následující regulární definicí:

$$\begin{aligned} \textit{letter} &\longrightarrow \texttt{[A - Za - z]} \\ \textit{digit} &\longrightarrow \texttt{[0 - 9]} \\ \textit{id} &\longrightarrow \textit{letter}(\textit{letter}|\textit{digit})^* \end{aligned}$$

■

**Příklad 2.5.** Čísla bez znaménka v Pascalu jsou řetězce jako 5280, 39.37, 6.336E4 nebo 1.894E-4. Následující regulární definice je přesnou specifikací této třídy řetězců:

$$
\begin{aligned}
digits &\longrightarrow [0-9]^+\\
optional_fraction &\longrightarrow (.\ digits)?\\
optional_exponent &\longrightarrow (\mathtt{E}\ (+|-)?\ digits)?\\
num &\longrightarrow digits\ optional_fraction\ optional_exponent
\end{aligned}
$$

Tato definice říká, že *optional_fraction* je buď desetinná tečka následovaná jednou nebo více číslicemi, nebo chybí (je to prázdný řetězec). *Optional_exponent*, pokud nechybí, je `E` následované volitelným `+` nebo `-` a jednou nebo více číslicemi. Povšimněte si, že za tečkou musí být alespoň jedna číslice, takže *num* neodpovídá řetězci `1.`, ale odpovídá řetězci `1.0`. ■

Regulární výrazy a regulární definice tvoří základní prostředek pro specifikaci lexikální struktury jazyka v systémech pro podporu návrhu překladačů, konkrétně v tzv. konstruktorech lexikálních analyzátorů. Na základě regulárních definic tyto konstruktory obvykle vytvoří odpovídající deterministický konečný automat reprezentovaný buď tabulkou přechodů a jejím interpretem nebo přímo programem realizujícím lexikální analýzu.

### 2.4.3 Konečné automaty

Dalším prostředkem, který lze využít jak pro specifikaci, tak i pro implementaci lexikálních analyzátorů, jsou konečné automaty. Pro naše potřeby vyjdeme z definice rozšířeného konečného automatu (viz [12]) jako pětice

$$(Q, \Sigma, f, q_0, F),$$

kde $Q$ je konečná množina vnitřních stavů, $\Sigma$ (neprázdná) vstupní abeceda, $f$ přechodová funkce $f : Q \times (\Sigma \cup \{\epsilon\}) -> 2^Q$, $q_0 \in Q$ počáteční stav a $F \subset Q$ množina koncových stavů.

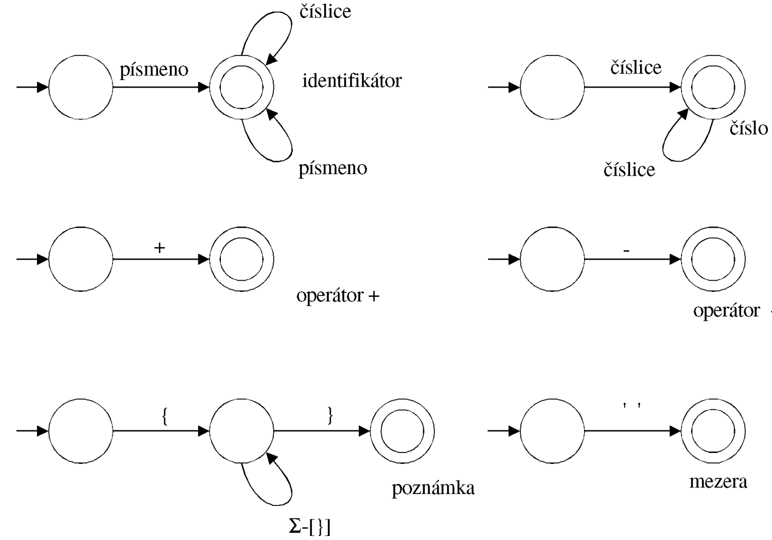
Pro všechny symboly jazyka můžeme sestrojit samostatné (obecně nedeterministické) automaty; všechny částečné automaty pak můžeme spojit do jediného automatu tak, že vytvoříme nový počáteční stav a pomocí $\epsilon$-přechodů jej propojíme s počátečními stavy jednotlivých výchozích automatů. Takto získaný automat pak převedeme na deterministický, např. algoritmem uvedeným v [13]. Dostaneme výsledný deterministický konečný automat, který pak můžeme implementovat některou z dále uvedených metod.

**Příklad 2.6.** Jazyk obsahující identifikátory, celá čísla bez znaménka, operátory '+' a '–', poznámky a mezery můžeme popsat částečnými automaty podle obr. 2.4. Výsledný automat, který získáme jejich spojením, je uveden na obr. 2.5.

## 2.5 Implementace lexikálního analyzátoru

Výběr konkrétní metody implementace je závislý spíše na tom, zda máme k dispozici a chceme použít nějaký konstruktor (v tom případě bude zřejmě nejvýhodnější popis regulárními výrazy) nebo zda budeme analyzátor psát přímo v některém programovacím jazyku; dalším kritériem (někdy i rozhodujícím) mohou být i požadavky na efektivitu lexikálního analyzátoru, neboť lexikální analyzátor zpracovává zdrojový program znak po znaku a často tedy přímo určuje rychlost celého překladu. Pro vlastní implementaci můžeme použít jednu z následujících metod:






Obrázek 2.4: Grafy částečných konečných automatů

1. přímá implementace s využitím všech prostředků, které poskytuje implementační jazyk,
2. implementace konečného automatu nebo
3. vytvoření analyzátoru konstruktorem.

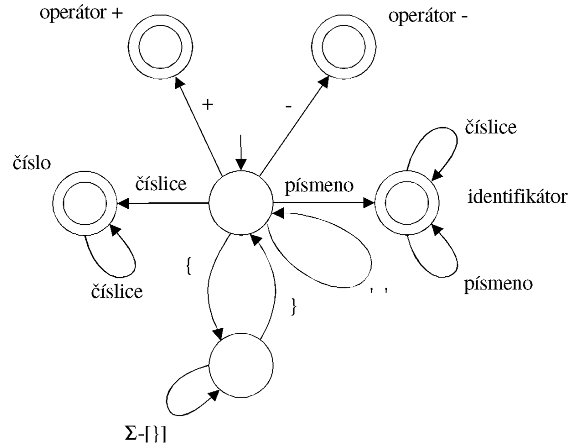
Z hlediska jednoduchosti je nejjednodušší použití konstruktoru a nejnáročnější přímá implementace, ovšem z hlediska efektivity překladu je pořadí obvykle přesně opačné. Z tohoto důvodu se často pro vývojové verze překladače použije konstruktoru, avšak pro definitivní překladač se lexikální analyzátor implementuje přímo.

V následujících odstavcích si předvedeme první dvě techniky na příkladu jazyka z obr. 2.5. Tento jazyk obsahuje identifikátory, celočíselné konstanty, operátory + a −, mezery a poznámky tvořené posloupností znaků uzavřených ve složených závorkách.

### 2.5.1 Přímá implementace

Přímá implementace lexikálního analyzátoru vychází z požadavků na maximální efektivitu jeho činnosti; využívá všech vhodných prostředků implementačního programovacího jazyka.

**Příklad 2.7.** Pro jazyk definovaný na obr. 2.5 ukážeme jednu z možných implementací lexikálního analyzátoru. Analyzátor bude představován funkcí `yylex` bez parametrů, která




Obrázek 2.5: Specifikace lexikálního analyzátoru pomocí DKA

po každém zavolání vrátí kód následujícího symbolu. V případě identifikátoru ponechá v proměnné `yytext` příslušný lexém a v proměnné `yyleng` jeho délku ve znacích, pro číselnou konstantu ponechá v proměnné `yyival` její binární hodnotu. (Použité názvy s výjimkou `yyival` jsou převzaty z pojmenování zavedeného v konstruktoru lex).

```
# include <stdio.h>
# include <ctype.h>
# define IDENT 256
# define NUM   257
char yytext[ 256 ];            /* lexém pro identifikátor */
int  yyleng;                   /* délka identifikátoru    */
int  yyival;                   /* hodnota čísla           */
int yylex(void)
{
    int ch;                    /* přečtený znak */
  START:
    while( (ch = getchar()) == ' ' );    /* vypuštění mezer */
    if ( isalpha(ch) ) {       /* zpracování identifikátoru */
       yyleng = 0;
       do {
          yytext[ yyleng++ ] = ch;
       } while ( isalnum(ch = getchar()) );
       yytext[ yyleng ] = '\0';
       ungetc( ch, stdin );   /* vrácení posledního znaku */
       return( IDENT );
```



```
    }
    else if ( isdigit(ch) ) {  /* zpracování čísla */
      yyival = 0;
      do {
         yyival = 10 * yyival + (ch - '0');
      } while ( isdigit(ch = getchar()) );
      ungetc( ch, stdin );   /* vrácení posledního znaku */
      return( NUM );
    }
    else if ( ch == '{' ) {    /* zpracování poznámky */
      while( (ch = getchar()) != '}' && ch != EOF );
      if ( ch == EOF ) {
         yyerror( "Neukončená poznámka" );
         return( EOF );
      }
      goto START;             /* pokračujeme dalším symbolem */
    }
    else                       /* ostatní znaky */
      return( ch );
}
```

Kódování symbolů je zvoleno tak, aby jednoznakové symboly mohly být reprezentovány přímo kódem odpovídajícího znaku. Složené symboly pak mají přiděleny kódy počínaje hodnotou 256. Analyzátor předává informaci o konci zdrojového souboru rovněž jako symbol — jeho kód (`EOF`) je převzat ze standardního záhlaví `<stdio.h>` jazyka C. Je-li na vstupu zjištěn znak, kterým nezačíná žádný z definovaných symbolů, je analyzátorem jeho kód vrácen a chyba není hlášena — ohlásí se až při syntaktické analýze (neboť vrácený kód nemůže odpovídat kódu žádného z očekávaných symbolů na vstupu). Rovněž by bylo možné zjistit, zda se jedná o znak '+' nebo '−' a v případě, že tomu tak není, nahlásit chybu (např. "Neplatný znak"), znak vynechat a pokračovat v analýze. ■

### 2.5.2 Implementace lexikálního analyzátoru jako automatu se stavovým řízením

V případě, že je lexikální struktura jazyka popsána konečným automatem, můžeme implementovat přímo činnost tohoto automatu, a to buď pomocí tabulky přechodové funkce automatu nebo přímo přepisem automatu do programu. Konstruktory lexikálních analyzátorů používají především první variantu, neboť jak formát tabulky, tak i příslušný interpretační program mohou být standardizovány.

**Příklad 2.8.** Ukážeme implementaci lexikálního analyzátoru pro jazyk z obr. 2.5 do programu v jazyce C formou automatu.

```
# include <stdio.h>
# include <ctype.h>

# define IDENT 256
# define NUM   257
```

```
char yytext[ 256 ];            /* lexém pro identifikátor */
int  yyleng;                   /* délka identifikátoru    */
int  yyival;                   /* hodnota čísla           */

static int state;              /* současný stav automatu */

int next( int newst )          /* přechod do nového stavu */
{
   state = newst;
   return getchar();
}

int yylex( void )
{
    int ch = next(0);          /* současný znak na vstupu */
    for ( ;; )
       switch (state) {
          case 0:       /* stav 0 - startovací stav */
             if ( ch == ' ' )
                ch = next(0);
             else if ( isalpha(ch) ) {
                yyleng = 1;
                yytext[ yyleng ] = ch;
                ch = next( 1 );
             }
             else if ( isdigit(ch) ) {
                yyival = ch - '0';
                ch = next( 2 );
             }
             else if ( ch == '{' )
                ch = next( 3 );
             else {
                return ( ch );
          case 1:       /* stav 1 - analýza identifikátoru */
             if( isalnum(ch) ) {
                yytext[ yyleng++ ] = ch;
                ch = next( 1 );
             }
             else {
                yytext[ yyleng ] = '\0';
                ungetc( ch, stdin );
                return( IDENT );
             }
          case 2:       /* stav 2 - analýza čísla */
             if( isdigit(ch) ) {
                yyival = 10 * yyival + (ch - '0');
                ch = next( 2 );
```



```
             }
             else {
                ungetc( ch, stdin );
                return( NUM );
             }
          case 3:       /* stav 3 - analýza poznámky */
             if( ch == EOF ) {
                yyerror( "Neukončená poznámka" );
                return( EOF );
             }
             else if( ch == '}' )
                ch = next( 0 );
             else
                ch = next( 3 );
       }
}
```

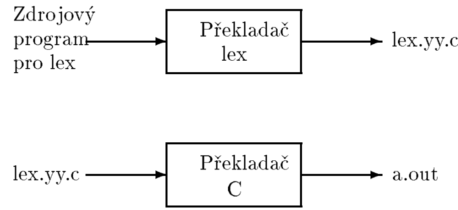
Volání funkce `next()` reprezentuje jednu hranu grafu přechodů; tato funkce nastaví nový stav automatu a přečte další znak ze vstupu. Povšimněte si, že uvedená implementace není zcela přesná, neboť zde nejsou realizovány koncové stavy reprezentující operátory '+' a '−'; v tomto smyslu se vlastně jedná o částečně optimalizovaný automat. I přesto je tato implementace mnohem méně efektivní než ta, která byla uvedena v předchozím odstavci. Je to způsobeno zejména neustálým rozhodováním o současném stavu a přechody, které nemění stav — např. ve stavu 0 při mezeře.

## 2.6 Lex — generátor lexikálních analyzátorů

### 2.6.1 Činnost programu `lex`

Pro vytváření lexikálních analyzátorů na základě speciálního zápisu založeného na regulárních výrazech bylo vytvořeno mnoho prostředků. S použitím regulárních výrazů a automatů pro specifikaci symbolů jsme se již seznámili. Nyní si uvedeme příklad prostředku, který by byl schopen vygenerovat lexikální analyzátor pouze na základě specifikace jazyka, konkrétně prostředek zvaný `lex`, který se široce využívá pro specifikaci lexikálních analyzátorů pro řadu jazyků. Budeme jej nazývat překladač `lex` a jeho vstupní specifikaci jazyk `lex`. Diskuse kolem tohoto jazyka nám umožní ukázat, jak lze specifikaci vzorů pomocí regulárních výrazů kombinovat s akcemi, tj. např. s vytvářením položek tabulky symbolů. Překladač `lex` byl implementován pod operačním systémem Unix (dále budeme popisovat právě tuto verzi), dnes je však dostupný i pod jinými operačními systémy, dokonce i v různých zdokonalených variantách.

`Lex` se obecně používá způsobem, který je znázorněn na obr. 2.6. Nejprve připravíme specifikaci lexikálního analyzátoru vytvořením zdrojového textu (např. v souboru `lex.l`) v jazyku `lex`. Potom soubor `lex.l` zpracujeme programem `lex` a tím vytvoříme program v C pod názvem `lex.yy.c`. Program `lex.yy.c` se skládá z tabulkové reprezentace grafu přechodů vytvořeného na základě regulárních výrazů obsažených v `lex.l` zároveň se standardními podprogramy, které tyto tabulky používají pro rozpoznávání symbolů. Akce spojené s regulárními výrazy v `lex.l` jsou reprezentovány úseky kódu v C; překladač `lex` je okopíruje přímo do




Obrázek 2.6: Vytvoření lexikálního analyzátoru programem `lex`

souboru `lex.yy.c`. Konečně se soubor `lex.yy.c` zpracuje překladačem cc jazyka C, který vytvoří modul lexikálního analyzátoru a případně jej i sestaví s ostatními moduly do cílového programu – překladače.

### 2.6.2 Struktura zdrojového textu

Program v jazyku `lex` se skládá ze tří částí, které jsou odděleny dvěma znaky `%` na začátku samostatného řádku:

```
deklarace
%%
překladová pravidla
%%
pomocné procedury
```

Oddíl deklarací obsahuje deklarace proměnných, pojmenovaných konstant a regulárních definicí. Deklarace, které se mají okopírovat do výstupního textu, musejí být uzavřeny do závorek `%{` a `%}`. Uvedené deklarace budou globální pro všechny funkce obsažené ve vygenerovaném programu. Regulární definice jsou příkazy ve tvaru

```
jméno   výraz
```

kde jméno je označení uvedeného regulárního výrazu, které může být v dalších výrazech použito ve tvaru {`jméno`}. Poznamenejme, že tyto definice jsou implementovány jako makra, takže případné chyby v jejich zápisu se projeví až při rozvoji v překladových pravidlech.

Druhý oddíl obsahuje vlastní definici lexikální struktury jazyka a činnosti analyzátoru formou překladových pravidel. Překladová pravidla pro `lex` jsou příkazy ve tvaru

$p_1$ $\quad$ $action_1$
$p_2$ $\quad$ $action_2$
...
$p_n$ $\quad$ $action_n$

kde $p_i$ jsou regulární výrazy a $action_i$ jsou části programu popisující činnost lexikálního analyzátoru po rozpoznání lexému odpovídajícího vzoru $p_i$ (jediný příkaz jazyka C nebo blok příkazů ve složených závorkách). V jazyku `lex` se akce zapisují jako příkazy jazyka C, obecně by však zde mohl být libovolný jiný implementační jazyk. Regulární výrazy jsou v pravidle zapsány bezprostředně od začátku řádku a bez mezer, od akce jsou odděleny alespoň jednou mezerou nebo tabulátorem.



| VÝRAZ | POPIS | PŘÍKLAD |
|---|---|---|
| `const` | const | `const` |
| $c$ | libovolný znak $c$, jež není operátorem | `a` |
| $\backslash c$ | libovolný znak $c$ | `*` |
| "$s$" | řetězec $s$ libovolných znaků | `"**"` |
| . | jakýkoliv znak kromě konce řádku | `a.*b` |
| ^ | začátek řádku | `^abc` |
| $ | konec řádku | `abc$` |
| $[s]$ | libovolný znak z množiny $s$ | `[abc]` |
| $[\hat{}s]$ | libovolný znak, který není v množině $s$ | `[^abc]` |
| $r*$ | 0 nebo více $r$ | `a*` |
| $r+$ | 1 nebo více $r$ | `a+` |
| $r?$ | 0 nebo jeden $r$ | `a?` |
| $r\{m,n\}$ | $m$ až $n$ výskytů $r$ | `a{1,5}` |
| $r_1r_2$ | $r_1$ následovaný $r_2$ | `ab` |
| $r_1 \mid r_2$ | $r_1$ nebo $r_2$ | `a\|b` |
| $(r)$ | $r$ | `(a\|b)` |
| $r_1/r_2$ | $r_1$, pokud za ním následuje $r_2$ | `abc/123` |

Obrázek 2.7: Regulární výrazy jazyka `lex`

Třetí oddíl obsahuje libovolné pomocné procedury potřebné pro akce; překladač `lex` pouze zkopíruje veškerý text ze třetího oddílu do výstupního souboru. Tyto procedury se mohou také překládat samostatně a potom spojit s lexikálním analyzátorem.

### 2.6.3 Zápis regulárních výrazů

Jazyk `lex` umožňuje podstatně komplikovanější zápis regulárních výrazů, jak ukazuje tabulka na obr. 2.7. Symbol $c$ v tabulce označuje jeden znak, $r$ regulární výraz a $s$ řetězec znaků. Zápis $\backslash c$, resp. "$s$" se používá tehdy, jestliže potřebujeme uvést některý ze speciálních znaků jazyka `lex` v jeho původním významu; jedná se o znaky `\ " . ^ $ [ ] * + ? { } |` a `/`. Zápis se zpětným lomítkem rovněž umožňuje zadat speciální řídicí znaky písmenem (např. `\t`, `\n`) nebo osmičkovým kódem (`\011`, `\015`).

### 2.6.4 Komunikace s okolím

Lexikální analyzátor vytvořený programem `lex` spolupracuje se syntaktickým analyzátorem následujícím způsobem. Po vyvolání funkce `yylex()` ze syntaktického analyzátoru začne lexikální analyzátor číst zbývající vstup po znacích až do okamžiku, kdy najde nejdelší prefix vstupního textu odpovídající jednomu z regulárních výrazů $p_i$. Potom provede akci $action_i$. Obvykle $action_i$ provede na konci příkaz `return( symbol )`, kterým vrátí řízení syntaktickému analyzátoru a zároveň předá kód přečteného symbolu. Pokud akce nekončí příkazem návratu, pokračuje lexikální analyzátor ve vyhledávání dalších symbolů až po dosažení akce, která způsobí návrat do syntaktického analyzátoru, nebo do nalezení konce vstupního souboru. Opakované vyhledávání lexémů až do explicitního návratu umožňuje lexikálnímu analyzátoru výhodně zpracovávat mezery a poznámky.

Lexikální analyzátor vrací syntaktickému analyzátoru jedinou hodnotu — kód symbolu. Pro předání hodnoty atributu s informacemi o lexému jsou k dispozici další proměnné:

```
int   yylineno;  /* číslo současného vstupního řádku  */
char  yytext[ ]; /* text naposledy přečteného lexému  */
int   yyleng;    /* délka naposledy přečteného lexému */
```

Navíc jsou zpřístupněny další proměnné, které umožňují měnit přiřazení vstupního a výstupního souboru pro lexikální analyzátor

```
FILE *yyin;      /* vstupní soubor  - implicitně stdin  */
FILE *yyout;     /* výstupní soubor - implicitně stdout */
```

Pro čtení jednoho znaku ze vstupního souboru a zápis jednoho znaku na výstup jsou k dispozici makra `input()` a `output()`, která je možno podle potřeby předefinovat. Výstupní soubor má význam tehdy, jestliže používáme `lex` pro vytvoření tzv. filtru, tj. programu, který čte vstupní soubor, provádí v něm určité transformace a transformovaný text zapisuje na výstup. Analyzátor vytvořený programem `lex` v případě, že část vstupního textu nelze přiřadit žádné z uvedených regulárních definic, tento text opíše do výstupního souboru `yyout`. Na to je třeba naopak pamatovat při návrhu skutečného lexikálního analyzátoru, kdy musí být pokryty skutečně všechny možné posloupnosti znaků na vstupu regulárními definicemi. Jinak by se např. v případě chybně zapsaného symbolu mohly na standardním výstupu objevit neočekávané texty.

**Příklad 2.9.** Poslední příklad této kapitoly ukazuje zápis lexikálního analyzátoru jazyka z obr. 2.5 prostředky konstruktoru `lex`.

```
%{
# include <stdlib.h>         /* pro funkci atoi() */
# define IDENT 256
# define NUM   257
int yyival;                  /* atribut symbolu NUM */
%}
                             /* regulární výrazy */
delim     [ \t\n]
ws        {delim}+|\{[^}]*\}
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+

%%
{ws}      {/* žádná akce a bez návratu */}
{id}      return(IDENT);
{number}  {yyival = atoi( yytext ); return(NUM);}
.         return(yytext[0]);
```

Obrázek 2.8: Program v jazyce `lex`

První výraz v části překladových pravidel udává, že po rozpoznání `ws`, tj. maximální posloupnosti mezer, tabelátorů, konců řádků a poznámek, se neprovede žádná akce a tedy že se bude



pokračovat čtením dalšího symbolu. V pravidle pro `id` obsahuje příslušná akce pouze návrat z lexikální analýzy a předání kódu symbolu `IDENT` jako návratové hodnoty. Pravidlo pro `number` nejprve převede textovou reprezentaci čísla z proměnné `yytext` na binární hodnotu standardní funkcí `atoi()` a vrátí kód symbolu `NUM`. Poznamenejme, že proměnná `yyival`, do níž se hodnota čísla ukládá, není definována programem `lex` a její definice tedy musí být uvedena v části deklarací. ■

**Příklad 2.10.** V příkladu 2.9 jsme si předvedli zápis lexikálního analyzátoru, u něhož jsme předpokládali opakované volání, vždy pro získání jediného vstupního symbolu. Pokud jiným způsobem se v jazyku `lex` vytvářejí filtry, které — jak jsme již uvedli — pouze transformují vstupní text a zapisují jej na výstup. Celá činnost filtru se tedy může provést v rámci jediného volání funkce `yylex()`. Následující program bude představovat filtr, který ze vstupního souboru vypustí všechny nadbytečné mezery a tabulátory.

```
%%
[ \t]+    { output(' '); }
```

Tato specifikace ze vstupního souboru vybírá pouze posloupnosti mezer a tabulátorů, které zkracuje na jedinou mezeru. Všechny ostatní znaky se přenášejí beze změny na výstup. ■

## 2.7 Zotavení po chybě v lexikální analýze

Přímo v lexikální analýze se rozpoznává pouze málo chyb, neboť lexikální analyzátor má na zdrojový program příliš omezený pohled. Pokud se ve zdrojovém programu v jazyce C objeví poprvé řetězec `fi` v kontextu

```
fi ( a == f(x) ) ...
```

nemůže lexikální analyzátor říci, zda `fi` je chybně napsané klíčové slovo `if` nebo nedeklarovaný identifikátor funkce. Vzhledem k tomu, že `fi` je platný identifikátor, musí lexikální analyzátor vrátit symbol pro identifikátor a nechat zpracování chyby na některé další fázi překladače.

Předpokládejme však, že se naskytla situace, ve které není lexikální analyzátor schopen pracovat, neboť žádný ze vzorů pro symboly neodpovídá prefixu zbývajícího vstupu. Snad nejsnadnější strategií zotavení je metoda, kdy ze zbývajícího vstupu vypouštíme znaky tak dlouho, až se lexikálnímu analyzátoru podaří rozpoznat další správně vytvořený symbol. Další možností je, že lexikální analyzátor, aniž nahlásí chybu, vrátí kód speciálního terminálního symbolu, který není obsažený v gramatice jazyka, a nechá hlášení chyby a zotavení na syntaktický analyzátor. Obě metody se v praxi běžně používají a jsou obvykle dostatečně účinné.

Jiné možné činnosti při zotavení z chyby jsou:

- vypuštění přebývajícího znaku,
- vložení chybějícího znaku,
- náhrada nesprávného znaku správným,
- vzájemná výměna dvou sousedních znaků.

Podobnými chybovými transformacemi se můžeme pokoušet opravit chybu. Nejjednodušší takovou strategií je zjišťování, zda se nedá použitím právě jedné transformace převést zbývající vstup na platný lexém. Tato strategie předpokládá, že většina lexikálních chyb je výsledkem jediné chybové transformace (např. překlepu při pořizování zdrojového textu); takový předpoklad obvykle (ale ne vždy) odpovídá praxi.

Jedním ze způsobů nalezení chyb v programu je výpočet minimálního počtu chybových transformací požadovaných pro převod chybného programu na syntakticky správný program. Říkáme, že chybný program obsahuje $k$ chyb, pokud nejkratší posloupnost chybových transformací, která jej zobrazuje na nějaký platný program, má délku $k$. Oprava chyb pomocí minimální vzdálenosti je vhodný teoretický nástroj, avšak v praxi se obecně nepoužívá pro její velmi náročnou implementaci. Několik experimentálních překladačů však používalo kritéria minimální vzdálenosti pro lokální opravy.

# Kapitola 3

# Syntaktická analýza

## 3.1 Činnost syntaktického analyzátoru

Během syntaktické analýzy se překladač snaží zjistit, zda zdrojový text tvoří větu odpovídající gramatice překládaného jazyka. K tomu využívá posloupnost lexikálních symbolů získanou jako výsledek lexikální analýzy. Pokud text obsahuje nějaké chyby, překladač je nahlásí a obvykle provede určité zotavení tak, aby i při výskytu chyb mohl pokračovat dále v činnosti a odhalit případné další chyby.

Při implementaci překladače se obvykle používá jednoho ze dvou základních přístupů — překladu *shora dolů* nebo *zdola nahoru*. Tyto názvy odpovídají postupu při vytváření derivačního stromu; při překladu shora dolů vycházíme ze startovacího symbolu gramatiky a snažíme se postupnou expanzí nonterminálních symbolů dospět až k terminálním symbolům odpovídajícím posloupnosti lexikálních symbolů na vstupu, při překladu zdola nahoru se naopak snažíme posloupnost terminálních symbolů ze vstupu redukovat až na startovací nonterminál. Uvedeným dvěma přístupům odpovídají také dvě základní třídy gramatik, konkrétně LL a LR gramatiky, které popisují určité dostatečně velké podmnožiny bezkontextových jazyků. Často se pro analyzátory implementované ručně využívá LL gramatik; analyzátory větší třídy LR jazyků se obvykle vytvářejí automatizovanými prostředky.

V praktické implementaci syntaktického analyzátoru obvykle požadujeme více než jenom informaci o syntaktické správnosti zdrojového programu. Výstupem analyzátoru bude určitá reprezentace zdrojového textu, která bude obsahovat pouze informace podstatné pro další průběh překladu. Touto reprezentací může být například derivační strom nebo obecně určitá posloupnost akcí, které vytvářejí vnitřní reprezentaci struktury zdrojového programu a uchovávají informace o sémantice těchto struktur (atributy — jména identifikátorů, hodnoty literálů apod.). Uchované informace pak využívá sémantická analýza pro vyhodnocení těch závislostí, které nelze popsat prostředky bezkontextových gramatik.

## 3.2 Syntaktická analýza shora dolů

V této části se budeme zabývat základními principy překladu shora dolů a implementací odpovídajícího syntaktického analyzátoru, nazývaného často prediktivní syntaktický analyzátor.

Překlad shora dolů můžeme popsat buď jako proces hledání levé derivace vstupního řetězce, nebo jako proces vytváření derivačního stromu počínaje jeho kořenem. Tento proces může být realizován obecně metodou "pokusu a omylu", kdy se snažíme v určitém bodě pře-





kladu aplikovat postupně jednotlivá pravidla gramatiky a v případě, že je aplikace určitého pravidla neúspěšná, provedeme návrat do bodu, ze kterého lze pokračovat dále volbou jiné varianty. Tento rekurzivní postup se nazývá *syntaktická analýza s návraty*; je značně neefektivní a pro účely překladu programovacích jazyků nevhodný. Naštěstí většina běžných konstrukcí programovacích jazyků je taková, že umožňují přímočarou analýzu bez návratů.

### 3.2.1 Množiny FIRST a FOLLOW

Konstrukce prediktivního analyzátoru je založena na dvou funkcích spojených s gramatikou $G$. Tyto funkce, $FIRST$ a $FOLLOW$, umožňují definovat řídicí tabulku pro deterministický zásobníkový automat. Množiny symbolů získané funkcí $FIRST$ i $FOLLOW$ lze také použít jako synchronizační množiny pro zotavení.

Je-li $\alpha$ řetězec symbolů gramatiky, potom $FIRST(\alpha)$ je množina terminálních symbolů, jimiž mohou začínat řetězce derivované z $\alpha$. Pokud $\alpha \overset{*}{\Rightarrow} \epsilon$, je $\epsilon$ rovněž ve $FIRST(\alpha)$.

Množinu $FOLLOW(A)$ pro nonterminál $A$ definujeme jako množinu všech terminálních symbolů $a$, které se mohou vyskytovat bezprostředně vpravo od $A$ v nějaké větné formě, tj. množina takových terminálních symbolů, pro něž existuje derivace ve tvaru $S \overset{*}{\Rightarrow} \alpha A a \beta$ pro nějaké $\alpha$ a $\beta$. Povšimněte si, že během derivace mohou mezi $A$ a $a$ být nějaké symboly. Pokud je tomu tak, pak tyto symboly derivují prázdný řetězec $\epsilon$ a vymizí. Může-li být $A$ nejpravějším symbolem v nějaké větné formě, je ve $FOLLOW(A)$ rovněž symbol \$, který představuje konec vstupního řetězce.

Množiny $FIRST(X)$ pro všechny symboly $X$ gramatiky vypočteme aplikací následujících pravidel opakovaně tak dlouho, až nelze do žádné množiny $FIRST$ přidat další terminální symbol nebo $\epsilon$.

1. Je-li $X$ terminální symbol, potom $FIRST(X)$ je rovno $\{X\}$.

2. Je-li $X \to \epsilon$ pravidlo, potom přidáme do $FIRST(X)$ symbol $\epsilon$.

3. Je-li $X$ nonterminál a $X \to Y_1 Y_2 \cdots Y_k$ pravidlo, potom přidáme do $FIRST(X)$ symbol $a$, jestliže pro nějaké $i$ je $a \in FIRST(Y_i)$ a $\epsilon$ je ve všech množinách $FIRST(Y_1)$, ..., $FIRST(Y_{i-1})$, tj. jestliže $Y_1 \cdots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$. Je-li $\epsilon$ ve $FIRST(Y_j)$ pro všechna $j = 1, 2, \ldots, k$, potom přidáme $\epsilon$ do $FIRST(X)$. Například všechny terminální symboly z $FIRST(Y_1)$ jsou určitě ve $FIRST(X)$. Pokud $Y_1$ nederivuje $\epsilon$, nepřidáme do $FIRST(X)$ již nic dalšího, ale jestliže $Y_1 \overset{*}{\Rightarrow} \epsilon$, přidáme $FIRST(Y_2)$ atd.

Nyní můžeme vypočítat $FIRST(X)$ pro libovolný řetězec $X_1 X_2 \cdots X_n$ následujícím postupem. Přidáme do $FIRST(X_1 X_2 \cdots X_n)$ všechny symboly z $FIRST(X_1)$ různé od $\epsilon$. Pokud je $\epsilon$ ve $FIRST(X_1)$, přidáme rovněž symboly z $FIRST(X_2)$, je-li $\epsilon$ ve $FIRST(X_1)$ i $FIRST(X_2)$, přidáme symboly z $FIRST(X_3)$ atd. Konečně přidáme do $FIRST(X_1 X_2 \cdots X_n)$ symbol $\epsilon$, pokud všechny množiny $FIRST(X_i)$ obsahují $\epsilon$.

Výpočet množin $FOLLOW(A)$ pro všechny nonterminály $A$ provedeme aplikací následujících pravidel opakovanou tak dlouho, až nelze do žádné množiny $FOLLOW$ přidat další symbol.

1. Do $FOLLOW(S)$, kde $S$ je startovací symbol gramatiky, vložíme symbol \$ označující konec vstupního řetězce.

2. Máme-li pravidlo $A \to \alpha B \beta$, potom vše z množiny $FIRST(\beta)$ kromě $\epsilon$ se umístí do $FOLLOW(B)$.

3. Máme-li pravidlo $A \to \alpha B$ nebo $A \to \alpha B \beta$ kde $FIRST(\beta)$ obsahuje $\epsilon$ (tj. $\beta \stackrel{*}{\Rightarrow} \epsilon$), potom prvky z množiny $FOLLOW(A)$ jsou obsaženy zároveň v množině $FOLLOW(B)$.

**Příklad 3.1.** Uvažujme gramatiku

$$\begin{array}{lrcl} (1) & E & \to & TE' \\ (2) & E' & \to & +TE' \\ (3) & & | & \epsilon \\ (4) & T & \to & FT' \\ (5) & T' & \to & *FT' \\ (6) & & | & \epsilon \\ (7) & F & \to & (E) \\ (8) & & | & \mathbf{id} \end{array}$$

Potom

$$\begin{array}{l} FIRST(E) = FIRST(T) = FIRST(F) = \{(, \mathbf{id}\} \\ FIRST(E') = \{+, \epsilon\} \\ FIRST(T') = \{*, \epsilon\} \\ FOLLOW(E) = FOLLOW(E') = \{), \$\} \\ FOLLOW(F) = \{+, *, ), \$\} \end{array}$$

Například **id** a levá závorka se přidaly do $FIRST(F)$ na základě pravidla (3) z definice $FIRST$ v obou případech s $i = 1$, neboť $FIRST(\mathbf{id}) = \{\mathbf{id}\}$ a $FIRST('(') = \{(\}$ podle pravidla (1). Potom podle pravidla (3) s $i = 1$ z pravidla $T \to FT'$ plyne, že **id** a levá závorka jsou rovněž ve $FIRST(T)$. Dále je například podle pravidla (2) symbol $\epsilon$ prvkem $FIRST(E')$.

Výpočet množin $FOLLOW$ zahájíme vložením \$ do $FOLLOW(E)$ podle pravidla (1). Podle (2) s pravidlem $F \to (E)$ je ve $FOLLOW(E)$ také pravá závorka. Aplikace (3) na pravidlo $E \to TE'$ vede k tomu, že \$ a pravá závorka jsou ve $FOLLOW(E')$. Vzhledem k tomu, že $E' \stackrel{*}{\Rightarrow} \epsilon$, jsou také ve $FOLLOW(T)$. Jako poslední příklad aplikace pravidel pro $FOLLOW$ uvažujme případ $T \to TE'$ v pravidle (2), podle něhož všechno z $FIRST(E')$ s výjimkou $\epsilon$ se musí umístit do $FOLLOW(T)$. To, že \$ je ve $FOLLOW(T)$, jsme již zjistili. ■

### 3.2.2 Konstrukce rozkladových tabulek

Syntaktický analyzátor pracující metodou shora dolů můžeme popsat jako zásobníkový automat tvořený vstupní páskou, zásobníkem, výstupní páskou a rozkladovou tabulkou. Automat čte symboly ze vstupní pásky a na výstupní pásku zapisuje čísla aplikovaných pravidel gramatiky — *levý rozklad* vstupní věty. Konfigurace tohoto automatu je dána trojicí

$$(x, X\alpha, \pi),$$

kde $x$ je nepřečtená část vstupního řetězce, $X\alpha$ obsah zásobníku (se symbolem $X$ na vrcholu) a $\pi$ je obsah výstupní pásky. Automat začíná pracovat v počáteční konfiguraci

$$(w, S\#, \epsilon),$$



kde $w$ je vstupní řetězec, $S$ startovací nonterminál a # speciální zásobníkový symbol označující dno zásobníku. Pokud automat přijme vstupní řetězec $w$, dostane se do koncové konfigurace

$$(\epsilon, \#, \pi),$$

kde $\pi$ je levý rozklad.

Rozkladová tabulka reprezentuje zobrazení

$$M : (\Sigma \cup N \cup \{\#\} \times (\Sigma \cup \{\$\}) \to \{\mathbf{expand}\ 1, \mathbf{expand}\ 2, \ldots, \mathbf{expand}\ n, \mathbf{pop}, \mathbf{accept}, \mathbf{error}\}$$

kde význam jednotlivých akcí je následující:

- **expand** $i$ Je-li $p_i : A \to \alpha$ $i$-té pravidlo gramatiky, na vrcholu zásobníku je nonterminál $A$, na vstupu symbol $a$ a $M[A, a] = \mathbf{expand}\ i$, provede automat přechod

  $$(ax, A\beta, \pi) \vdash (ax, \alpha\beta, \pi\, i)$$

  tj. nonterminál $A$ se na vrcholu zásobníku nahradí pravou stranou $\alpha$ pravidla $p_i$ a na výstup se dá číslo použitého pravidla $i$.

- **pop** Je-li na vstupu i na vrcholu zásobníku týž terminální symbol $a$, provede automat přechod

  $$(ax, a\beta, \pi) \vdash (x, \beta, \pi)$$

  tj. symbol $a$ se odstraní z vrcholu zásobníku i ze vstupu.

- **accept** Akce **accept** představuje přijetí vstupního řetězce v koncové konfiguraci automatu, přičemž výstupní řetězec obsahuje úplný levý rozklad vstupní věty.

- **error** Akce **error** nastane tehdy, jestliže vstupní řetězec není prvkem jazyka, takže automat nemůže dále pokračovat v činnosti.

**Příklad 3.2.** Rozkladová tabulka deterministického zásobníkového automatu pro gramatiku

$$\begin{array}{lrcl} (1) & S & \to & aAS \\ (2) & S & \to & b \\ (3) & A & \to & a \\ (4) & A & \to & bSA \end{array}$$

bude mít následující tvar (akce **expand** $i$ je zapsána jako **e**$i$, akce **accept** jako **acc** a prázdná políčka představují akci **error**):

| ZÁSOBNÍK | VSTUPNÍ SYMBOL | | |
|---|---|---|---|
| | $a$ | $b$ | \$ |
| $S$ | **e**1 | **e**2 | |
| $A$ | **e**3 | **e**4 | |
| $a$ | **pop** | | |
| $b$ | | **pop** | |
| # | | | **acc** |

Pro vstupní řetězec *abbab* potom můžeme vytvořit následující posloupnost přechodů automatu:

$$(abbab\$, S\#, \epsilon) \overset{e1}{\vdash} (abbab\$, aAS\#, 1) \overset{pop}{\vdash} (bbab\$, AS\#, 1) \overset{e4}{\vdash} (bbab\$, bSAS\#, 14) \overset{pop}{\vdash}$$
$$(bab, SAS\#, 14) \overset{e2}{\vdash} (bab\$, bAS\#, 142) \overset{pop}{\vdash} (ab\$, AS\#, 142) \overset{e3}{\vdash} (ab\$, aS\#, 1423) \overset{pop}{\vdash}$$
$$(b\$, S\#, 1423) \overset{e2}{\vdash} (b\$, b\#, 14232) \overset{pop}{\vdash} (\$, \#, 14232),$$

která nám dá rozklad věty *abbab* ve tvaru 14232. ■

Pro konstrukci rozkladové tabulky deterministického zásobníkového automatu ke gramatice $G$ můžeme využít algoritmu 3.1. Je založen na následující myšlence. Předpokládejme, že $A \to \alpha$ je pravidlo a že $a$ je ve $FIRST(\alpha)$. Potom, je-li současným vstupním symbolem $a$, provede analyzátor expanzi $A$ na $\alpha$. Jediná komplikace nastane, pokud $\alpha = \epsilon$ nebo $\alpha \overset{*}{\Rightarrow} \epsilon$. V tom případě musíme opět expandovat $A$ na $\alpha$, je-li současný vstupní symbol ve $FOLLOW(A)$ nebo byl-li dosažen konec vstupního řetězce (symbol \$) a \$ je ve $FOLLOW(A)$. Akce **pop** se bude provádět tehdy, je-li na vrcholu zásobníku i na vstupu týž terminální symbol a akce **accept** nastane v situaci, kdy bude vstupní řetězec vyčerpán (na vstupu bude ukončovací symbol \$) a zásobník vyprázdněn (na vrcholu bude symbol #).

**Algoritmus 3.1.** (Konstrukce rozkladové tabulky prediktivního analyzátoru)
*Vstup.* Gramatika $G$.
*Výstup.* Rozkladová tabulka $M$.
*Metoda.*

1. Pro všechna pravidla $p_i$ tvaru $A \to \alpha$ proveď kroky 2 a 3.
2. Pro všechny terminální symboly $a$ ve $FIRST(\alpha)$ přidej **expand** $i$ do $M[A, a]$.
3. Je-li $\epsilon$ ve $FIRST(\alpha)$, přidej **expand** $i$ do $M[A, b]$ pro všechny terminální symboly $b$ z množiny $FOLLOW(A)$. Pokud $\epsilon$ je ve $FIRST(\alpha)$ a \$ ve $FOLLOW(A)$, přidej **expand** $i$ do $M[A, \$]$.
4. Pro všechny terminální symboly $a$ přidej **pop** do $M[a, a]$.
5. Nastav $M[\#, \$]$ na **pop**.
6. Všechny nedefinované položky v $M$ nastav na **error**.

**Příklad 3.3.** Použijme algoritmus 3.1 na gramatiku z příkladu 3.1. Vzhledem k tomu, že $FIRST(TE') = FIRST(T) = \{(, \mathbf{id}\}$, budou položky $M[E, (]$ a $M[E, \mathbf{id}]$ obsahovat **expand** 1.

Pravidlo $E' \to +TE'$ vede k tomu, že $M[E', +]$ bude obsahovat **expand** 2. Pravidlo $E' \to \epsilon$ vede dále k tomu, že $M[E', )]$ a $M[E', \$]$ budou obsahovat **expand** 3, neboť $FOLLOW(E') = \{), \$\}$.

Celá rozkladová tabulka vytvořená algoritmem 3.1 je na obr. 3.1.

### 3.2.3 LL(1) gramatiky

Algoritmus 3.1 lze aplikovat na libovolnou gramatiku $G$ a získat tak rozkladovou tabulku $M$. Pro některé gramatiky se však může stát, že v některých položkách rozkladové tabulky budeme mít více konfliktních akcí. Například je-li gramatika $G$ zleva rekurzivní nebo nejednoznačná, bude tabulka $M$ obsahovat alespoň jednu násobně definovanou položku.



| ZÁSOBNÍK | VSTUPNÍ SYMBOL | | | | | |
|---|---|---|---|---|---|---|
| | **id** | + | * | ( | ) | \$ |
| $E$ | e1 | | | e1 | | |
| $E'$ | | e2 | | | e3 | e3 |
| $T$ | e4 | | | e4 | | |
| $T'$ | | e6 | e5 | | e6 | e6 |
| $F$ | e8 | | | e7 | | |
| **id** | **pop** | | | | | |
| + | | **pop** | | | | |
| * | | | **pop** | | | |
| ( | | | | **pop** | | |
| ) | | | | | **pop** | |
| \$ | | | | | | **acc** |

Obrázek 3.1: Rozkladová tabulka prediktivního analyzátoru

Gramatika, jejíž rozkladová tabulka neobsahuje násobně definované položky, se nazývá *LL(1) gramatika*. První "L" v názvu znamená, že se vstupní text prohlíží zleva doprava, druhé "L" představuje vytváření levého rozkladu a "1" vyjadřuje počet symbolů ve vstupním textu, které potřebujeme znát při rozhodování o průběhu analýzy. Lze ukázat, že algoritmus 3.1 pro všechny LL(1) gramatiky $G$ vede k rozkladové tabulce deterministického zásobníkového automatu, který přijímá právě jazyk $L(G)$.

Z definice LL(1) gramatiky (viz [12]) vyplývá několik vlastností, které umožňují rozhodnout, zda daná gramatika je či není typu LL(1). Následující dvě vlastnosti musí každá LL(1) gramatika nutně splňovat:

Nechť $A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n$ jsou všechna $A$-pravidla gramatiky $G$. Potom:

- *Vlastnost FF.* Množiny $FIRST$ všech pravých stran musejí být po dvojicích disjunktní, tj.
$$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset \text{ pro } i \neq j$$
- *Vlastnost FFL.* Je-li dále pro nějaké $i$ $\alpha_i \overset{*}{\Rightarrow} \epsilon$, musí být $FOLLOW(A)$ po dvojicích disjunktní s množinami $FIRST$ zbývajících pravých stran, tj.
$$FIRST(\alpha_j) \cap FOLLOW(A) = \emptyset \text{ pro } i \neq j$$

Z uvedených pravidel například vyplývá, že LL(1) gramatika nemůže obsahovat levou rekurzi, neboť by pro některý nonterminál $A$ takový, že $A \overset{+}{\Rightarrow} A\alpha, \alpha \in (N \cup \Sigma)^*$, byla porušena podmínka FF. Například je-li v gramatice přímo pravidlo $A \to A\alpha | \beta$, potom $FIRST(\beta) \subseteq FIRST(A\alpha)$.

### 3.2.4 Transformace na LL(1) gramatiku

V mnoha případech není výchozí gramatika, pro kterou chceme vytvořit syntaktický analyzátor, typu LL(1). To znamená, že v ní existují pravidla, která porušují některou z podmínek FF nebo FFL. Transformaci takové gramatiky na typ LL(1) nám mohou umožnit následující postupy (podrobnější popis uveden v [12]):

- *Odstranění levé rekurze* Jak již bylo uvedeno výše, gramatika, která obsahuje levou rekurzi, nemůže být typu LL(1). Obecně můžeme zleva rekurzivní pravidlo zapsat jako

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \cdots \mid \beta_m$$

kde řetězce $\beta_i$ nezačínají nonterminálem $A$. Takové pravidlo můžeme přepsat zavedením nového nonterminálu $A'$ jako

$$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_m A'$$

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A' \mid \epsilon$$

- *Faktorizace pravidel* Začíná-li několik pravých stran $A$-pravidla týmž řetězcem terminálních symbolů, tj. má-li pravidlo tvar

$$A \to \beta\alpha_1 \mid \beta\alpha_2 \mid \cdots \mid \beta\alpha_n,$$

můžeme provést jejich "vytknutí" opět zavedením nového nonterminálu $A'$ s pravidly

$$A \to \beta A'$$

$$A' \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$$

Tato úprava, stejně jako předchozí, však nemusí zaručit, že nepřinese další konflikty. Budou-li například některé z řetězců $\alpha_i$ neprázdný průnik množin $FIRST$, dojde opět k porušení podmínky FF v nonterminálu $A'$.

- *Eliminace pravidel* Některým konfliktům se můžeme vyhnout tak, že za některé nonterminály dosadíme jejich pravé strany a tím odstraníme z gramatiky pravidla, která způsobovala konflikt.

- *Redukce množiny FOLLOW* Je-li pro některý nonterminál porušena podmínka FFL, můžeme přidat nový nonterminál, který vede ke zmenšení počtu prvků konfliktní množiny $FOLLOW$ a případně i k disjunktnosti této množiny $FOLLOW$ s množinami $FIRST$ zbývajících pravých stran pravidel konfliktního nonterminálu (příklad viz [12], str. 103).

Uvedené transformace nemusí obecně vést k cíli, a to i v případě, že k transformované gramatice LL(1) gramatika existuje.

### 3.2.5 Analýza rekurzivním sestupem

Jednou z implementací syntaktické analýzy shora dolů je analýza rekurzivním sestupem. Tato metoda spočívá v zápisu samostatných procedur pro analýzu každého nonterminálního symbolu gramatiky. Překlad programu se pak spustí voláním procedury odpovídající startovacímu nonterminálu.

Máme-li pro nonterminál $A$ jediné pravidlo ve tvaru $A \to X_1\ X_2\ \ldots\ X_n$, bude tělo příslušné procedury obsahovat posloupnost akcí provádějících postupně analýzu symbolů $X_1$ až $X_n$. Je-li symbol $X_i$ nonterminálním symbolem gramatiky, bude odpovídající akcí volání podprogramu pro analýzu symbolu $X_i$, je-li $X_i$ terminální symbol, zavoláme podprogram `expect(`$X_i$`)`. Tento podprogram zjistí, zda je na vstupu požadovaný symbol a v případě, že



```
procedure expect(s: symbol);
begin
   if sym = s then
      lex
   else
      error
end;
```

Obrázek 3.2: Implementace procedury `expect`

ano, přečte další vstupní symbol; v opačném případě nahlásí syntaktickou chybu. Příklad implementace procedury `expect` v jazyce Pascal je na obr. 3.2. Předpokládáme, že lexikální analyzátor je reprezentován procedurou `lex`, která při každém zavolání naplní globální proměnnou `sym` typu `symbol` následujícím vstupním symbolem.

Například pro analýzu nonterminálu $A$ s jediným pravidlem $A \to xBy$ bude implementace procedury následující (předpokládáme, že terminálním symbolům $x$ a $y$ odpovídají konstanty `SYM_X` a `SYM_Y`):

```
procedure A;
begin
   expect(SYM_X);
   B;
   expectSYM_Y)
end;
```

V případě, že nonterminál $A$ je definován více $A$-pravidly gramatiky, např. pokud gramatika obsahuje $A$-pravidla $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$, musíme nejprve na základě následujícího vstupního symbolu vybrat vhodnou pravou stranu. Pro každou variantu $\alpha_i$ budeme mít úsek ve tvaru

```
if sym in Φ(A, αi) then begin
   /* implementace analýzy řetězce αi */
end
```

kde funkce $\Phi(A, \alpha_i)$ je definována jako

$$\Phi(A,\alpha) = \begin{cases} FIRST(\alpha), & \epsilon \notin FIRST(\alpha) \\ FOLLOW(A) \cup (FIRST(\alpha) \setminus \{\epsilon\}), & \epsilon \in FIRST(\alpha) \end{cases}$$

Tato funkce definuje množinu symbolů, které se mohou vyskytovat na vstupu v okamžiku expanze nonterminálu $A$ na řetězec $\alpha$. Pokud tento řetězec vždy obsahuje alespoň jeden symbol, je touto množinou $FIRST(\alpha)$. Může-li však expandovaný řetězec být prázdný, je třeba očekávat na vstupu i ty symboly, které jsou součástí množiny $FOLLOW(A)$ nonterminálu na levé straně pravidla. Je-li na vstupu symbol, který nepatří do žádné z množin $\Phi(A, \alpha_i)$, jde o syntaktickou chybu.

Vzhledem k tomu, že výběr pravé strany musí být u analyzátoru bez návratů jednoznačný, musí být množiny symbolů definované funkcí $\Phi(A, \alpha_i)$ pro jednotlivé pravé strany $\alpha_i$ disjunktní. Toto tvrzení ale není nic jiného, než vyjádření podmínek FF a FFL pro LL(1) gramatiku.

**Příklad 3.4.** Mějme dánu gramatiku pro aritmetický výraz s operátory `+` a `*`, závorkami a celočíselnými konstantami:

$$
\begin{aligned}
E &\rightarrow T\ E1 \\
E1 &\rightarrow +\ T\ E1 \mid \epsilon \\
T &\rightarrow F\ T1 \\
T1 &\rightarrow *\ F\ T1 \mid \epsilon \\
F &\rightarrow (\ E\ ) \mid \mathbf{id}
\end{aligned}
$$

Pro nonterminál $E1$ můžeme vypočítat následující množiny:

$$
\begin{aligned}
FIRST(+\ T\ E1) &= \{+\}, \\
FIRST(\epsilon) &= \{\epsilon\}, \\
FOLLOW(E1) &= \{), \$\}, \\
\Phi(E1, +\ T\ E1) &= \{+\}, \\
\Phi(E1, \epsilon) &= \{), \$\},
\end{aligned}
$$

takže jej můžeme implementovat procedurou

```
procedure E1;
begin
    if sym in [ADDSYM] then begin
        expect(ADDSYM);
        T;
        E1
        end
    else if sym in [RPRSYM, EOFSYM] then begin
        /* prázdná pravá strana */
        end
    else
        error
end;
```

Typ `symbol` je v tomto případě reprezentován výčtem konstant `ADDSYM` (operátor $+$), `MULSYM` (operátor $*$), `LPRSYM` (levá závorka), `RPRSYM` (pravá závorka), `IDSYM` (identifikátor) a `EOFSYM` (konec vstupního textu \$).

Je zřejmé, že uvedené řešení lze implementovat mnohem efektivněji, pokud provedeme následující optimalizace:

- Test, zda je symbol obsažen v jednoprvkové množině, lze nahradit přímo testem na rovnost.
- V případě, že pravá strana pravidla začíná terminálním symbolem, není třeba volat proceduru `expect`, neboť máme již při výběru pravé strany zaručen kladný výsledek testu na obr. 3.2. Můžeme tedy rovnou volat lexikální analyzátor.
- Je-li pravá strana pravidla prázdná (tj. je-li tvořena pouze symbolem $\epsilon$), je možné ji implementovat vždy jako poslední a obrátit příslušný test.



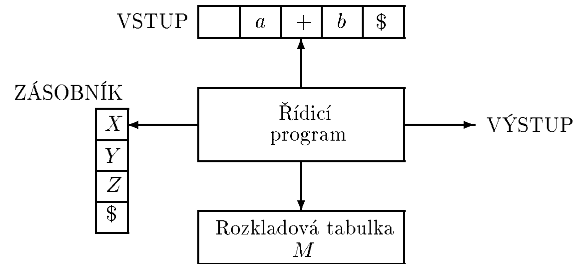
Po naznačených optimalizacích dostaneme konečnou verzi procedury analyzující nonterminál $E1$:

```
procedure E1;
begin
    if sym = ADDSYM then begin
        lex;
        T;
        E1
        end
    else if not (sym in [RPRSYM, EOFSYM]) then
        error;
end;
```

Podobným způsobem můžeme implementovat i zbývající nonterminály gramatiky. ■

### 3.2.6 Nerekurzivní prediktivní analýza

Implementace syntaktického analyzátoru z předchozího článku využívala pro uchování informací o rozpracované části věty implicitního zásobníku, který používá hostitelský překladač (tj. v našem případě překladač Pascalu) pro realizaci volání rekurzivních podprogramů. Je však také možné vytvořit prediktivní syntaktický analyzátor, který používá svůj vlastní zásobník. Struktura takového analyzátoru je na obr. 3.3.




Obrázek 3.3: Model nerekurzivního prediktivního syntaktického analyzátoru

Tento typ analyzátoru, nazývaný *syntaktický analyzátor řízený tabulkou*, je tvořen vstupní pamětí, zásobníkem, rozkladovou tabulkou a výstupem. Vstupní paměť obsahuje analyzovaný řetězec zakončený speciálním symbolem \$, který označuje konec vstupního řetězce. Zásobník obsahuje posloupnost symbolů gramatiky; dno zásobníku je indikováno opět speciálním symbolem #. Rozkladová tabulka je dvojrozměrné pole $M[A, a]$, kde $A$ je nonterminál a $a$ je terminální symbol nebo symbol \$.

Samostatnou částí analyzátoru je řídicí program, který opakovaně prohlíží symbol $X$ na vrcholu zásobníku a současný vstupní symbol $a$, na základě nichž se rozhoduje o své další činnosti. Algoritmus rozhodování je následující:

- Je-li $X = \#$ a $a = \$$, vyčerpali jsme vstupní řetězec i zásobník; analyzátor se zastaví a ohlásí úspěšné ukončení.

- Je-li $X = a \neq \$$, odstraníme symbol $X$ z vrcholu zásobníku a přesuneme se na následující vstupní symbol.
- Je-li $X$ nonterminální symbol, provedeme jeho expanzi na některou z odpovídajících pravých stran pravidel gramatiky. Pokud položka rozkladové tabulky $M[X, a]$ obsahuje $X$-pravidlo gramatiky, nahradíme symbol $X$ na vrcholu zásobníku pravou stranou tohoto pravidla a na výstup předáme číslo použitého pravidla. Pokud je však $M[X, a] =$ **error**, jde o syntaktickou chybu, kterou musí analyzátor nahlásit a provést zotavení.
- V ostatních případech jde opět o syntaktickou chybu.

Tento algoritmus můžeme vyjádřit programem na obr. 3.4. Proměnná *top* obsahuje index vrcholu zásobníku symbolů *stack*, funkce *pop*() odstraní vrchol zásobníku a funkce *push*() uloží na zásobník řetězec symbolů. Funkce *error*() provádí hlášení syntaktických chyb a případné zotavení, funkce *lex*() představuje lexikální analyzátor, který při každém zavolání vrátí jeden symbol ze vstupu.

```
top := 0;
push(#S);
a := lex();
repeat
    X := stack[top];
    if X je terminální symbol nebo $ then
        if X = a then begin
            pop();
            a := lex();
        end
        else error()
    else    /* X je nonterminál */
        if M[X, a] = X → Y1Y2···Yk then begin
            pop();
            push(YkYk−1···Y1);
            vypiš číslo použitého pravidla
        end
        else error()
until X = #     /* zásobník je prázdný */
```

Obrázek 3.4: Řídicí program prediktivního analyzátoru

### 3.2.7 Zotavení po chybě při analýze shora dolů

K důležitým úkolům syntaktického analyzátoru patří také diagnostická činnost. Aby v rámci jednoho průchodu zdrojovým programem kompilátor odhalil co nejvíce chyb, je třeba implementovat prostředky, které dovolí, aby syntaktický analyzátor pokračoval v kontrole správnosti programu i po výskytu syntaktické chyby. Problém zotavení ze syntaktické chyby není obecně jednoduchý. Běžně používané metody vycházejí z následujícího obecného postupu:



1. Po odhalení syntaktické chyby se ve vstupním řetězci hledá místo (*bod zotavení*, od kterého může analýza pokračovat v činnosti, přičemž se vynechá určitá část textu. Bod zotavení je obvykle dán nalezením symbolu z množiny tzv. *klíčů*.
2. Syntaktický analyzátor provede synchronizaci podle pozice nalezeného klíče v gramatice a pokračuje dále v činnosti.

Množina klíčů musí být definována tak, aby obsahovala pokud možno pouze ty symboly, jejichž výskyt v gramatice je jednoznačný. Tím lze zajistit vyšší spolehlivost synchronizace analyzátoru při zotavování. Například v gramatice jazyka Pascal je klíčové slovo **else** použito jednoznačně, na rozdíl od identifikátoru nebo klíčového slova **end** (konec složeného příkazu, příkazu **case**, resp. záznamu). Je-li však množina klíčů příliš omezená, roste délka neanalyzovaného textu, který se vynechává při vyhledávání klíče ve vstupní větě.

Pro zotavení na základě množiny klíčů se používajím například tyto metody:

- *Nerekurzivní metoda s pevnou množinou klíčů.* Tato metoda vychází z předem vypočtené množiny klíčů. Ke každému klíči je k dispozici informace o tom, kterou syntaktickou konstrukci ukončuje. Například klíč `')'` může ukončovat výrazy a klíč `';'` příkazy. Vyskytne-li se pak chyba během analýzy výrazu a při zotavení se najde pravá závorka, odstraní se ze zásobníku všechno, co souviselo s rozpracovaným výrazem a pokračuje se v analýze tak, jako by byl výraz analyzován správně.
- *Rekurzivní metoda s pevnou množinou klíčů.* Předchozí metoda se dá vylepšit ještě tím, že se určí rovněž množiny klíčů, kterými začínají jisté syntaktické konstrukce. Je-li během vyhledávání bodu zotavení nalezen některý z těchto klíčů, spustí se analýza vnořené konstrukce a po jejím ukončení se pokračuje v zotavení. Tím je možné omezit rozsah neanalyzovaného textu a mohou být tedy odhaleny další chyby v zanořených konstrukcích.
- *Metoda s dynamicky budovanou množinou klíčů.* Při této metodě se množina klíčů vytváří vždy na základě okamžitého kontextu; například při analýze příkazů v těle pascalovského cyklu **repeat** bude klíčem symbol **until**, zatímco při analýze výrazu v indexu bude klíčem pravá závorka. Jednou z metod této skupiny je Hartmannova metoda, která jako množiny klíčů využívá sjednocení množin *FOLLOW* rozpracovaných nonterminálů. Její implementací se budeme dále zabývat podrobněji.

#### Hartmannovo schéma zotavení

Každému syntakticky správně vytvořenému programu analyzovanému syntaktickým analyzátorem přísluší derivační strom. Při analýze metodou rekurzivního sestupu je derivační strom budován postupným vyvoláváním procedur odpovídajících jednotlivým nonterminálům gramatiky a jejich prováděním. Výskyt syntaktické chyby představuje z hlediska syntaktického analyzátoru situaci, kdy v jistém stadiu rozpracování derivačního stromu není možné v budování tohoto stromu pokračovat. Začlenění prostředků pro zotavení po chybě Hartmannovou metodou předpokládá, že analyzátor při výskytu chyby

- ukončí vytváření derivačního podstromu obsahujícího chybu (neurčujeme zatím, kterého podstromu; v nejhorším případě dojde k ukončení vytváření celého stromu a tím i analýzy) s tím, že tento podstrom je nadále uvažován jako správně vytvořený

- přeskočí všechny symboly na vstupu mezi chybou a koncem fráze odpovídající uzavřenému derivačnímu podstromu.

Snahou dobrého zotavování je uzavřít po chybě co nejtěsnější podstrom obklopující chybu (podstrom, jehož kořen je co nejvíce vzdálen od vrcholu derivačního stromu). Čím těsnější podstrom je uzavřen, tím méně symbolů je třeba přeskočit. Přeskakované symboly nejsou analyzovány; mohou být zdrojem dalších syntaktických chyb a pokud je symbolů přeskočeno příliš mnoho, nelze v jedné analýze odhalit všechny chyby.

V každém okamžiku analýzy je vytvářen derivační podstrom pro jistý počet nonterminálů, přičemž tyto podstromy jsou do sebe vnořeny. Přiřaďme každému rozpracovanému derivačnímu podstromu množinu symbolů nazvanou $CONTEXT(A)$, která je sjednocením množin $FOLLOW(A_i)$ všech nonterminálů, jež mají v okamžiku expanze nonterminálu $A$ rozpracovaný derivační podstrom, včetně množiny $FOLLOW(A)$. Vznikne-li v průběhu vytváření derivačního podstromu pro nonterminál $A$ chyba, musí proběhnout zotavení.

Množina $CONTEXT(A)$ je dynamicky budovanou množinou klíčů, které využíváme při hledání bodu zotavení. Přeskočení symbolů na vstupu mezi chybou a koncem fráze odpovídající jistému podstromu, je realizováno přeskočením všech symbolů na vstupu, které nejsou v množině $CONTEXT(A)$. Protože všechny symboly z množiny $CONTEXT(A)$ jsou zároveň prvky jedné nebo více množin $FOLLOW$ pro jednotlivé vnořené derivační podstromy, je zajištěno, že bude přeskočen nejmenší možný počet symbolů ze vstupu a nalezen nejbližší možný bod zotavení v daném kontextu. Zároveň je třeba postupně uzavřít analýzu všech nonterminálů počínaje od nejvnořenějšího, v jejichž množinách $FOLLOW$ není obsažen nastavený vstupní symbol. Posledním nonterminálem, jehož analýza se uzavře, je nonterminál, v jehož množině $FOLLOW$ bod zotavení je.

Během analýzy metodou rekurzivního sestupu může dojít k detekci syntaktické chyby ve dvou situacích:

- je-li na vstupu jiný terminální symbol než se očekává, nebo
- nelze-li při expanzi nonterminálu vybrat na základě současného vstupního symbolu žádnou pravou stranu pravidla (vstupní symbol není prvkem $\Phi(A, \alpha_i)$ pro žádné $i$).

První případ odpovídá situaci, kdy se chyba hlásí z procedury `expect`, druhý případ nastává bezprostředně při vstupu do procedury analyzující konkrétní nonterminál. Je-li k dispozici množina klíčů $CONTEXT$ (budeme ji nazývat také *kontextová množina*, neboť definuje kontext, v němž analýza probíhá), můžeme upravit proceduru `expect` tak, aby při chybě provedla zároveň i synchronizaci, jak ukazuje obr. 3.5.

Test na začátku analýzy nonterminálu zároveň se zotavením může provést procedura `check(s, context)`, která jako první parametr obdrží sjednocení množin $\Phi(A, \alpha_i)$ pro všechny pravé strany $\alpha_i$ nonterminálu $A$. Není-li současný vstupní symbol v této množině, nahlásí se chyba a provede se zotavení pomocí kontextové množiny. Při hledání bodu zotavení se ještě připouští, aby se na vstupu ještě objevil symbol z množiny očekávaných symbolů $s$, což umožňuje efektivní zotavení v situaci, kdy je na vstupu nějaký symbol navíc.

Vlastní postup při začlenění zotavení do analýzy rekurzivním sestupem je pak následující:

- procedury pro analýzu nonterminálů budou jako vstupní parametr předávaný hodnotou dostávat aktuální kontextovou množinu, tj. deklarace procedur budou mít tvar

```
procedure A(context: symbols);
```



```
type symbols = set of symbol;

procedure expect(s:symbol; context: symbols);
begin
   if sym = s then
      lex
   else begin
      error;
      while not (sym in context) do lex
   end
end;

procedure check(s, c: symbols);
begin
   if not (sym in s) then begin
      error;
      while not (sym in c+s) do lex
   end
end;
```

Obrázek 3.5: Implementace pomocných procedur pro zotavení

- při volání procedury pro analýzu nonterminálu nebo procedury `expect` se vždy vypočte nová kontextová množina
- před volbou varianty v nonterminálu se zavolá procedura `check(`$\bigcup_i \Phi(A, \alpha_i)$`, context)`, která zjistí, zda současný vstupní symbol odpovídá některé z pravých stran pro nonterminál $A$.

Výpočet kontextové množiny symbolu $X_i$ na pravé straně pravidla $A \to X_1X_2 \ldots X_iX_{i+1} \ldots X_k$ spočívá v rozšíření současné kontextové množiny $CONTEXT(A)$ o symboly, které se stanou klíči pro analyzovaný terminální nebo nonterminální symbol. Možné jsou například tyto přístupy:

1. Kontextovou množinu nonterminálu $X_i$ vždy rozšíříme o prvky množiny $FOLLOW(X_i)$, tj.

   $$CONTEXT(X_i) = CONTEXT(A) \cup FOLLOW(X_i), X_i \in N$$

   zatímco kontextovou množinu terminálních symbolů (tj. argument procedury *expect*) ponecháme původní, tj.

   $$CONTEXT(X_i) = CONTEXT(A), X_i \in \Sigma$$

2. Kontextovou množinu symbolu $X_i$ (terminálního i nonterminálního) vždy rozšíříme o symboly, jimiž může začínat zbývající část řetězce na pravé straně pravidla, tj.

   $$CONTEXT(X_i) = CONTEXT(A) \cup (FIRST(X_{i+1} \ldots X_k) \setminus \{\epsilon\})$$

3. Kontextovou množinu symbolu $X_i$ rozšíříme o symboly ležící ve $FIRST$ všech následujících symbolů v pravidle, tj.

$$CONTEXT(X_i) = CONTEXT(A) \cup ( \bigcup_{j=i+1}^{k} FIRST(X_j) \setminus \{\epsilon\})$$

První varianta je nejjednodušší, ovšem vyžaduje výpočet množin $FOLLOW$ a vede obecně k přeskočení zbytku rozpracovaného pravidla při chybě uvnitř některého ze symbolů na pravé straně. Další dvě varianty se liší mohutností kontextové množiny, přičemž nejvýhodnější řešení je zřejmě kombinací všech tří přístupů, kdy do kontextové množiny nebudeme přidávat ty symboly, které jsou nejednoznačné (tj. takové, které se ve zdrojovém textu mohou vyskytovat v různých významech). Na výběru kontextových množin podstatně závisí kvalita zotavení, která se projevuje nejen počtem odhalených skutečných chyb, ale (v opačném smyslu) i počtem hlášených zavlečených chyb

**Příklad 3.5.** Uvažujme následující gramatiku pro deklarace proměnných s inicializací:

$$\begin{aligned} S &\rightarrow \textbf{var id } L = \textbf{num} \\ L &\rightarrow \textbf{, id } L \mid \epsilon \end{aligned}$$

Použijeme-li posledního přístupu k výpočtu kontextových množin, můžeme syntaktickou analýzu se zotavením implementovat následujícími procedurami (symboly **var**, **id**, **num**, čárka, rovnítko a $ jsou pojmenovány po řadě `VARSYM`, `IDSYM`, `NUMSYM`, `COMSYM`, `EQSYM` a `EOFSYM`):

```
procedure S(c: symbols);
begin
   expect(VARSYM, c + [IDSYM, COMSYM, EQSYM, NUMSYM]);
   expect(IDSYM,  c + [COMSYM, EQSYM, NUMSYM]);
   L(c + [EQSYM, NUMSYM]);
   expect(EQSYM,  c + [NUMSYM]);
   expect(NUMSYM, c)
end;
procedure L(c: symbols);
begin
   check([COMSYM, EQSYM], c);
   if sym = COMSYM then begin
      lex;
      expect(IDSYM, c + [COMSYM]);
      L(c)
   end
end;
```

Poznamenejme, že v situaci, kdy některý nonterminál $A$ může generovat prázdný řetězec, je podle definice funkce $\Phi$ součástí prvního parametru funkce `check` také množina $FOLLOW(A)$. Vzhledem k tomu, jak se vytváří kontextová množina, můžeme množinu $FOLLOW(A)$ nahradit obecnější množinou $CONTEXT(A)$, která v dané situaci lépe reprezentuje množinu přípustných symbolů, které mohou v konkrétní situaci za nonterminálem $A$ následovat. Například při analýze výrazů reprezentovaných nonterminálem $E$ je v množině $FOLLOW(E)$ vždy obsažena pravá závorka jako důsledek pravidla $E \rightarrow (E)$. Pokud však



nebyla ještě otevřena žádná levá závorka, není pravá závorka vlastně platným klíčem a neměla by být ani součástí kontextové množiny. Například v proceduře pro nonterminál $L$ se může funkce `check` volat jako

```
check(c + [COMSYM], c)
```

■

# Kapitola 4

# Syntaxí řízený překlad

## 4.1 Základní pojmy teorie překladu

V tomto odstavci zavedeme některé základní pojmy teorie překladu, na které dále navážeme definicemi pojmů, které se přímo využívají při implementaci překladače.

**Definice 4.1.** Nechť $\Sigma$ a $\Delta$ jsou abecedy. Abecedu $\Sigma$ nazvěme *vstupní abecedou*, $\Delta$ *výstupní abecedou*. *Překladem* jazyka $L_1 \subset \Sigma^*$ do jazyka $L_2 \subset \Delta^*$ nazveme relaci TRAN : $L_1 \to L_2$. Je-li $[x, y] \in$ TRAN, pak řetězec $y$ nazýváme *výstupem* pro řetězec $x$. ■

Typickým příkladem překladu je překlad infixového zápisu aritmetického výrazu na postfixový. Tento překlad je nekonečný (relace TRAN obsahuje nekonečně mnoho dvojic řetězců) a relace, jež ho definuje, je ve skutečnosti funkcí, neboť ke každému infixovému zápisu výrazu existuje právě jeden zápis postfixový. Problém konečné specifikace nekonečného překladu je analogický specifikaci nekonečného jazyka. Stejně jako tomu bylo u syntaktické analýzy, jsou i zde dva možné přístupy — prostřednictvím generativního systému (gramatiky) nebo prostřednictvím automatu.

Generativní systém, nazývaný *překladová párová gramatika*, je založený na dvou vzájemně spojených bezkontextových gramatikách. První z nich, tzv. *vstupní gramatika*, popisuje jazyk tvořený všemi větami zdrojového jazyka $L_1$; druhá, výstupní gramatika, popisuje jazyk $L_2 = \{y | [x, y] \in \text{TRAN}\}$ tvořený všemi výstupy pro řetězce jazyka $L_1$. Mechanismus zobecněné derivace umožňuje paralelní derivaci řetězce $x$ ve vstupní gramatice a řetězce $y$ ve výstupní gramatice.

Druhý přístup ke specifikaci překladu využívá pojmu *překladový automat*, který je získán rozšířením konečného nebo zásobníkového automatu o výstupní pásku a výstupní funkci, která předepisuje výstup automatu. Překlad definovaný překladovým automatem je množina dvojic řetězců $[x, y]$ takových, že automat přijme řetězec $x$ a na výstup vyšle řetězec $y$. Teorií překladových automatů se tento učební text nebude zabývat, případné zájemce odkazujeme na [2].

**Definice 4.2.** *Překladová párová gramatika* je pětice

$$V = (N, \Sigma, \Delta, P, S)$$

kde $N$ je konečná množina *nonterminálních symbolů*, $\Sigma$ konečná *vstupní abeceda*, $\Delta$ konečná *výstupní abeceda*, $P$ množina *přepisovacích pravidel* a $S \in N$ *startovací* (nonterminální) *sym-*





*bol*. Pravidla mají tvar $A \longrightarrow \alpha, \beta$, kde $A \in N$, $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Delta)^*$ a nonterminály v řetězci $\beta$ jsou permutací nonterminálů z řetězce $\alpha$. ■

Překladová párová gramatika představuje nejobecnější specifikaci překladu; z hlediska implementace přináší značné komplikace možnost záměny pořadí nonterminálů na pravé straně pravidel. V případě, že tuto možnost zakážeme, můžeme vstupní a výstupní část pravidla sloučit do jediného řetězce (za předpokladu, že jsou vstupní i výstupní abecedy disjunktní). Tím se dostáváme k pojmu *překladová gramatika*, který pro nás bude východiskem při dalším popisu činnosti překladače.

**Definice 4.3.** Nechť $V = (N, \Sigma, \Delta, P, S)$ je překladová párová gramatika, přičemž $N \cap \Delta = \emptyset$ a množina $P$ obsahuje pouze pravidla tvaru

$$A \longrightarrow x_0 B_1 x_1 B_2 \ldots B_k x_k, y_0 B_1 y_1 B_2 \ldots B_k y_k \tag{4.1}$$

pro $x_i \in \Sigma^*$, $y_i \in \Delta^*$, $0 \le i \le k$. Pak *překladová gramatika* $G_V$ příslušející gramatice $V$ je pětice $G_V = (N, \Sigma, \Delta, P', S)$, kde množina $P'$ obsahuje pouze pravidla ve tvaru

$$A \longrightarrow x_0 y_0 B_1 x_1 y_1 B_2 \ldots B_k x_k y_k$$

odvozená z původních pravidel ve tvaru (4.1). ■

**Příklad 4.1.** Uvažujme překladovou párovou gramatiku

$$V = (\{E, T, F\}, \{+, *, i, (, )\}, \{+, *, i\}, P, E)$$

s pravidly

$$\begin{aligned}
E &\to E + T,\ E\ T\ + \\
E &\to T,\ T \\
T &\to T * F,\ T\ F\ * \\
T &\to F,\ F \\
F &\to (\ E\ ),\ E \\
F &\to i,\ i
\end{aligned}$$

Tato párová gramatika generuje překlad infixového aritmetického výrazu do postfixového výrazu, např.

$$\begin{aligned}
[E, E] &\Rightarrow [E + T, ET+] \Rightarrow [T + T, TT+] \Rightarrow [F + T, FT+] \Rightarrow \\
&\Rightarrow [i + T, iT+] \Rightarrow [i + T * F, iTF * +] \Rightarrow [i + F * F, iFF * =] \Rightarrow \\
&\Rightarrow [i + i * F, iiF * +] \Rightarrow [i + i * i, iii * +]
\end{aligned}$$

■

**Příklad 4.2.** Pravidla gramatiky z předchozího příkladu zachovávají pořadí odpovídajících si nonterminálů na pravých stranách. Po přejmenování symbolů výstupní abecedy tedy můžeme odvodit následující překladovou gramatiku:

$$G_V = (\{E, T, F\}, \{+, *, i, (, )\}, \{\texttt{ADD}, \texttt{MUL}, \texttt{ID}\}, P', E)$$

s pravidly

$$\begin{aligned} E &\rightarrow E + T\ \texttt{ADD} \\ E &\rightarrow T \\ T &\rightarrow T * F\ \texttt{MUL} \\ T &\rightarrow F \\ F &\rightarrow (\ E\ ) \\ F &\rightarrow i\ \texttt{ID} \end{aligned}$$

Tato gramatika umožňuje provést např. následující derivaci:

$$\begin{aligned} E &\Rightarrow E + T\ \texttt{ADD} \Rightarrow T + T\ \texttt{ADD} \Rightarrow F + T\ \texttt{ADD} \Rightarrow i\ ID + T\ \texttt{ADD} \Rightarrow \\ &\Rightarrow i\ \texttt{ID} + T * F\ \texttt{MUL ADD} \Rightarrow i\ \texttt{ID} + F * F\ \texttt{MUL ADD} \Rightarrow \\ &\Rightarrow i\ \texttt{ID} + i\ \texttt{ID} * F\ \texttt{MUL ADD} \Rightarrow i\ \texttt{ID} + i\ \texttt{ID} * i\ \texttt{ID MUL ADD} \end{aligned}$$

Vidíme, že ve větě "`i ID + i ID * i ID MUL ADD`" tvoří symboly vstupní abecedy jak jdou po sobě vstup a výstupní symboly odpovídají výstupu překladu, tj. dvojice `(i+i*i, ID ID ID MUL ADD)` je prvek překladu. ■

Z hlediska implementace mohou být symboly výstupní abecedy reprezentovány jako skutečné výstupní symboly (symboly `ID`, `ADD` a `MUL` z předchozích příkladů by např. mohly představovat instrukce zásobníkového mezikódu pro vyhodnocení aritmetického výrazu) nebo jako akce, např. pro symbol `ADD` volání procedury pro vygenerování instrukce sčítání nebo dokonce pro provedení součtu v případě interpretačního překladače. V dalších odstavcích se budeme zabývat rozšířením pojmu překladové gramatiky o atributy symbolů, přičemž konkrétní reprezentaci jednotlivých symbolů nebudeme v definicích uvažovat.

## 4.2 Atributovaný překlad

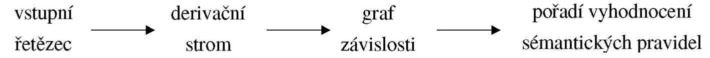
Prozatím jsme se zabývali pouze kontrolou, zda je věta, kterou překládáme, prvkem překládaného jazyka. V této kapitole přiřadíme k syntaktickým konstrukcím další informace — atributy, které se vyhodnocují na základě sémantických pravidel.

Pro připojení sémantických pravidel k pravidlům gramatiky existují dvě notace, syntaxí řízené definice a překladová schemata. Syntaxí řízené definice jsou specifikací překladu na vysoké úrovni abstrakce. Ukrývají mnoho implementačních detailů a osvobozují uživatele od nutnosti specifikovat explicitně pořadí, v jakém se bude překlad provádět. Překladová schemata určují pořadí vyhodnocování sémantických pravidel, takže umožňují ukázat i některé implementační detaily.

Obecně jak při překladu pomocí syntaxí řízených definic, tak i při použití překladových schemat rozkládáme vstupní posloupnost symbolů, budujeme derivační strom a potom procházíme stromem tak, abychom vyhodnotili sémantická pravidla v uzlech derivačního stromu (viz obr. 4.1). Vyhodnocením sémantických pravidel může být generování kódu, ukládání informací do tabulky symbolů, vydávání zpráv o chybách nebo provádění nějakých jiných činností. Výsledkem vyhodnocení sémantických pravidel je překlad posloupnosti vstupních symbolů.

Implementace nemusí být doslova shodná se schematem na obr. 4.1. Speciální případy syntaxí řízeného překladu lze implementovat v jednom průchodu s vyhodnocením sémantických






Obrázek 4.1: Celkový pohled na syntaxí řízený překlad

pravidel během analýzy, bez explicitní konstrukce derivačního stromu nebo grafu ukazujícího závislosti mezi atributy. Vzhledem k tomu, že jednoprůchodová implementace je důležitá pro efektivitu překladače, je velká část této kapitoly věnována studiu takových případů. Jedna důležitá podtřída, zvaná *L-atributové definice*, zahrnuje téměř všechny překlady, které lze provádět bez explicitní konstrukce derivačního stromu.

### 4.2.1 Atributové překladové gramatiky

**Definice 4.4.** *Atributová překladová gramatika* (APG) je trojice

$$G_{AP} = (G_P, A, F),$$

kde $G_P = (N, \Sigma, \Delta, R, S)$ je překladová gramatika, $A$ množina *atributů* a $F$ množina *sémantických pravidel*. V případě, že je množina výstupních symbolů $\Delta$ prázdná, hovoříme pouze o *atributové gramatice* (AG).

Pro každý symbol $X \in N \cup \Sigma \cup \Delta$ jsou dány dvě (případně prázdné) disjunktní množiny — množina $I(X)$ *dědičných atributů* a množina $S(X)$ *syntetizovaných atributů*, přičemž pro $a \in I(S)$ jsou zadány počáteční hodnoty (dědičné atributy startovacího nonterminálu) a pro terminální symboly $X \in \Sigma$ je $I(X) = \emptyset$ (terminální symboly nemají dědičné atributy) a jejich syntetizované atributy jsou zadány.

Nechť $r$-té pravidlo gramatiky má tvar i$X_0 \longrightarrow X_1 X_2 \ldots X_{n_r}$, kde $X_0 \in N$, $X_i \in N \cup \Sigma \cup \Delta$ pro $1 \leq i \leq n_r$. Pak

a) pro každý symbol $X_k$, $1 \leq k \leq n_r$ na pravé straně pravidla $r$ a jeho dědičný atribut $d \in I(X_k)$ je dáno sémantické pravidlo

$$d = f_r^{d,k}(a_1, a_2, \ldots, a_n)$$

kde $a_i$, $1 \leq i \leq n$ jsou atributy symbolů v témže pravidle $r$,

b) pro každý syntetizovaný atribut $s$ symbolu $X_0$ na levé straně pravidla $r$ je dáno sémantické pravidlo

$$s = f_r^{s,0}(a_1, a_2, \ldots, a_n)$$

kde $a_i$, $1 \leq i \leq n$ jsou atributy symbolů v témže pravidle $r$, a

c) pro každý syntetizovaný atribut výstupního symbolu $X_k \in \Delta$ v pravidle $r$ je dáno sémantické pravidlo

$$s = f_r^{s,k}(a_1, a_2, \ldots, a_n)$$

kde $a_i$, $1 \leq i \leq n$ jsou pouze dědičné atributy symbolu $X_k \in \Delta$. ■

| PRAVIDLA | SÉMANTICKÁ PRAVIDLA |
|---|---|
| $E_0 \longrightarrow E_1 + T$ | $E_0.val = E_1.val + T.val$ |
| $E \longrightarrow T$ | $E.val = T.val$ |
| $T_0 \longrightarrow T_1 * F$ | $T_0.val = T_1.val * F.val$ |
| $T \longrightarrow F$ | $T.val = F.val$ |
| $F \longrightarrow (E)$ | $F.val = E.val$ |
| $F \longrightarrow \mathbf{num}$ | $F.val = \mathbf{num}.ival$ |

Obrázek 4.2: Atributová gramatika pro aritmetický výraz

Sémantická pravidla realizujeme obvykle příkazy (funkcemi) vhodného vyššího programovacího jazyka (např. C nebo Pascal). Atributy pak chápeme jako proměnné či parametry jistého datového typu.

V dalším textu budeme atributy symbolů pojmenovávat kvalifikovanými jmény ve tvaru $X.a$, kde $X$ je jméno symbolu a $a$ jméno atributu. Sémantické funkce budeme psát vždy za pravidlo gramatiky, k němuž se vztahují. V případě, že se v jednom pravidle bude vyskytovat určitý symbol vícekrát, rozlišíme jednotlivé výskyty pomocí indexu.

**Příklad 4.3.** Atributová gramatika na obr. 4.2 popisuje aritmetický výraz tvořený celočíselnými konstantami, operátory `+`, `*` a závorkami. Nonterminály $E$, $T$ a $F$ mají celočíselný syntetizovaný atribut *val*, který udává hodnotu příslušných podvýrazů, syntetizovaný atribut *ival* terminálního symbolu **num** udává hodnotu celočíselné konstanty získanou z lexikální analýzy. Jednotlivá sémantická pravidla počítají hodnotu atributu *val* nonterminálu na levé straně z hodnot *val* symbolů na pravé straně pravidel gramatiky. ■

Vyhodnocením sémantického pravidla definujeme hodnoty atributů uzlů derivačního stromu pro vstupní řetězec. Derivační strom s hodnotami atributů v každém uzlu nazýváme ohodnocený derivační strom. Proces výpočtu hodnot atributů v uzlech nazýváme ohodnocením derivačního stromu.

**Příklad 4.4.** Atributová gramatika z příkladu 4.3 vypočte hodnotu aritmetického výrazu s desítkovými čísly, závorkami a operátory `+` a `*`. Například pro výraz `3*5+4` vypočte hodnotu 19 jako hodnotu atributu $E.val$ startovacího nonterminálu $E$. Obr. 4.3 obsahuje ohodnocený derivační strom pro vstup `3*5+4`.

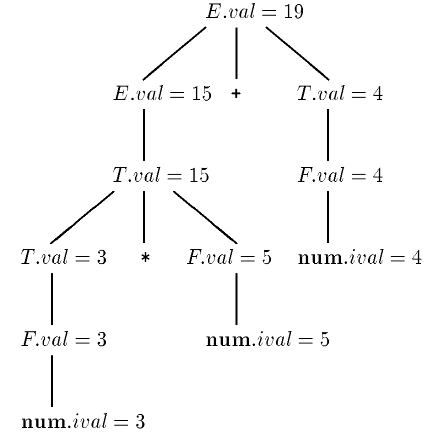
Abychom ukázali, jak se atributy vyhodnocují, uvažujme levý dolní vnitřní uzel, odpovídající použití pravidla $F \longrightarrow \mathbf{num}$. Odpovídající sémantické pravidlo $F.val := \mathbf{num}.ival$ přidělí atributu $F.val$ v tomto uzlu hodnotu 3, neboť hodnota $\mathbf{num}.ival$ následníka uzlu je 3. Podobně v předchůdci tohoto $F$-uzlu má atribut $T.val$ hodnotu 3. Nyní uvažujme uzel pro pravidlo $T \longrightarrow T * F$. Hodnota atributu $T.val$ v tomto uzlu je definována jako

| PRAVIDLO | SÉMANTICKÉ PRAVIDLO |
|---|---|
| $T_0 \longrightarrow T_1 * F$ | $T_0.val := T_1.val * F.val$ |

Pokud aplikujeme na tento uzel uvedené sémantické pravidlo, bude mít $T_1.val$ hodnotu 3 levého následníka a $F.val$ hodnotu 5 pravého následníka. $T_0.val$ tedy dostane v tomto uzlu hodnotu 15. Konečně pro startovací nonterminál $E$ se podobným způsobem vypočte hodnota 19. ■

Sémantické funkce z definice atributové gramatiky nám z matematického hlediska umožňují pouze vyhodnocovat atributy a předávat je mezi jednotlivými symboly gramatiky






Obrázek 4.3: Ohodnocený derivační strom pro `3*5+4`

bez možnosti využití vedlejších efektů (např. výstupní operace, práce s globálními proměnnými apod.). Pokud připustíme, aby sémantické funkce měly vedlejší efekty, hovoříme o *syntaxí řízené definici* (SDD).

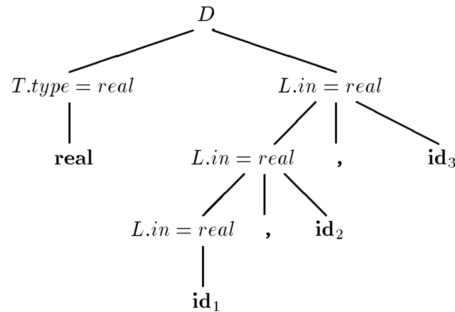
**Příklad 4.5.** Deklarace generovaná nonterminálem $D$ v syntaxí řízené definici na obr. 4.4 se skládá z klíčového slova **int** nebo **real** následovaného seznamem identifikátorů. Nonterminál $T$ má syntetizovaný atribut *type*, jehož hodnota je určena klíčovým slovem v deklaraci. Sémantické pravidlo $L.in := T.type$, svázané s pravidlem $D \longrightarrow T\ L$, nastavuje dědičný atribut $L.in$ na hodnotu typu v deklaraci. Pravidla přenášejí tento typ dolů derivačním stromem pomocí dědičného atributu $L.in$. Pravidla spojená s pravidlem gramatiky pro $L$ volají proceduru *addtype*, která připojí typ k položce tabulky symbolů pro každý identifikátor (na položku ukazuje atribut *entry*).

Obr. 4.5 ukazuje ohodnocený derivační strom pro větu **real** $\mathbf{id}_1, \mathbf{id}_2, \mathbf{id}_3$. Hodnota $L.in$ ve třech $L$-uzlech udává typ identifikátorů $\mathbf{id}_1$, $\mathbf{id}_2$ a $\mathbf{id}_3$. Tyto hodnoty se určí výpočtem hodnoty atributu $T.type$ levého následníka kořene a pak výpočtem $L.in$ shora dolů ve třech $L$-uzlech pravého podstromu kořene. V každém $L$-uzlu také voláme proceduru *addtype*, která

| PRAVIDLO | SÉMANTICKÉ PRAVIDLO |
|---|---|
| $D \longrightarrow T\ L$ | $L.in := T.type$ |
| $T \longrightarrow \mathbf{int}$ | $T.type := integer$ |
| $T \longrightarrow \mathbf{real}$ | $T.type := real$ |
| $L \longrightarrow L_1, \mathbf{id}$ | $L_1.in := L.in$ |
| | $addtype(\mathbf{id}.entry, L.in)$ |
| $L \longrightarrow \mathbf{id}$ | $addtype(\mathbf{id}.entry, L.in)$ |

Obrázek 4.4: Syntaxí řízená definice s dědičným atributem $L.in$

uloží do tabulky symbolů informaci o tom, že identifikátor v pravém podstromu uzlu má typ *real*. ■




Obrázek 4.5: Derivační strom s dědičnými atributy v uzlech $L$

### 4.2.2 Graf závislosti

Sémantická pravidla udávají závislosti mezi atributy. Tyto závislosti reprezentujeme *grafem závislosti* (dependency graph), ze kterého pak můžeme odvodit pořadí vyhodnocení sémantických pravidel. Závisí-li atribut $b$ uzlu derivačního stromu na atributu $c$, pak musí být sémantické pravidlo pro $b$ vyhodnoceno po sémantickém pravidle definujícím $c$.

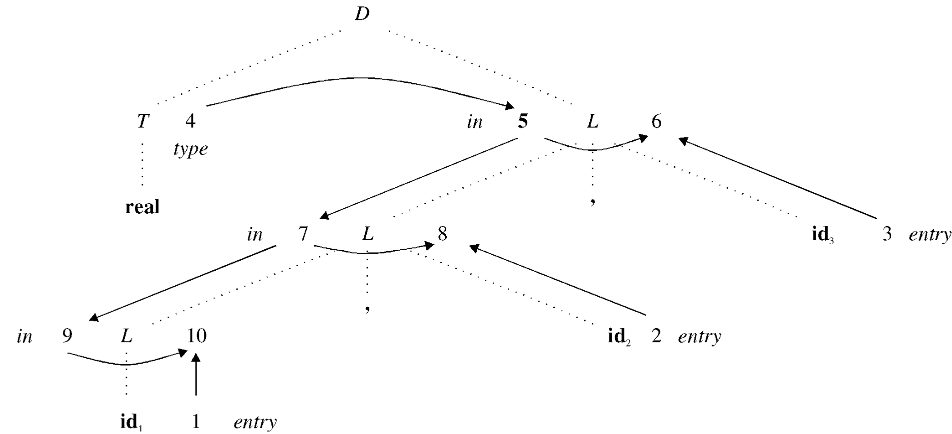
Ještě před tím, než začneme konstruovat graf závislosti k danému derivačnímu stromu, převedeme všechna sémantická pravidla do tvaru $b := f(c_1, c_2, \dots, c_k)$ zavedením prázdného syntetizovaného atributu $b$ pro všechna sémantická pravidla tvořená voláním procedury. Graf obsahuje ke každému atributu jeden uzel a hrany vedoucí z uzlu $b$ do uzlu $c$, pokud atribut $b$ závisí na atributu $c$. Graf závislosti lze konkrétně vytvořit následujícím způsobem:

**for** každý uzel $n$ derivačního stromu **do**
  **for** každý atribut a symbolu gramatiky v $n$ **do**
    vytvoř uzel grafu závislosti pro $a$;
**for** každý uzel $n$ derivačního stromu **do**
  **for** každé sémantické pravidlo $b := f(c_1, c_2, \dots, c_k)$
      spojené s pravidlem použitým v $n$ **do**
    **for** $i := 1$ **to** $k$ **do**
      vytvoř hranu z uzlu pro $c_i$ do uzlu pro $b$;

Předpokládejme například, že $A.a := f(X.x, Y.y)$ je sémantické pravidlo k pravidlu gramatiky $A \longrightarrow X\ Y$. Toto pravidlo definuje syntetizovaný atribut $A.a$, jež závisí na atributech $X.x$ a $Y.y$. Je-li takové pravidlo použito v derivačním stromu, budeme mít v grafu závislosti tři uzly $A.a$, $X.x$ a $Y.y$ s hranou vedoucí z $A.a$ do $X.x$ ($A.a$ závisí na $X.x$) a hranou z $A.a$ do $Y.y$ ($A.a$ závisí také na $Y.y$).

Je-li s pravidlem $A \longrightarrow X\ Y$ spojeno sémantické pravidlo $X.i := g(A.a, Y.y)$, bude graf závislosti obsahovat hranu do $A.a$ z $X.i$ a také do $A.a$ z $Y.y$, neboť $X.i$ závisí jak na $A.a$, tak na $Y.y$.






Obrázek 4.6: Graf závislosti

**Příklad 4.6.** Obr. 4.6 ukazuje graf závislosti pro derivační strom na obr. 4.5. Uzly v grafu závislosti jsou označeny čísly; tato čísla budeme používat dále. Pro $L.in$ zde máme hranu vedoucí do uzlu 5 z uzlu 4 pro $T.type$, neboť dědičný atribut $L.in$ závisí na atributu $T.type$ na základě sémantického pravidla $L_1.in := T.type$ pro pravidlo gramatiky $D \longrightarrow T\ L$. Dvě hrany vedoucí dolů do uzlů 7 a 9 vyplývají ze závislosti $L_1.in$ na $L.in$ podle sémantického pravidla $L_1.in := L.in$ pro pravidlo gramatiky $L \longrightarrow L_1, \mathbf{id}$. Sémantické pravidlo $addtype(\mathbf{id}.entry, L.in)$ spojené s $L$-pravidly vede k vytvoření prázdného atributu. Uzly 6, 8 a 10 byly vytvořeny právě pro tyto prázdné atributy. ■

### 4.2.3 Pořadí vyhodnocení pravidel

**Definice 4.5.** *Topologický sort* orientovaného acyklického grafu je libovolné uspořádání $m_1, m_2, \dots, m_k$ uzlů grafu takové, že hrany vedou z uzlů uvedených dříve do uzlů uvedených později; to znamená, že je-li $m_i \longrightarrow m_j$ hrana z $m_i$ do $m_j$, potom se $m_i$ vyskytuje v tomto uspořádání před $m_j$. ■

Libovolný topologický sort grafu závislosti je použitelný jako pořadí, v němž se mají vyhodnocovat sémantická pravidla spojená s uzly derivačního stromu. V topologickém sortu jsou závislé atributy $c_1, c_2, \dots, c_k$ v sémantickém pravidle $b := f(c_1, c_2, \dots, c_k)$ k dispozici ještě před vyhodnocením $f$.

Překlad specifikovaný syntaxí řízenou definicí můžeme provést následujícím způsobem. Pro vytvoření derivačního stromu k zadanému vstupu použijeme výchozí gramatiku. Podle předchozího algoritmu vytvoříme graf závislosti. Z topologického sortu grafu závislosti získáme pořadí vyhodnocení sémantických pravidel a vyhodnocením sémantických pravidel v tomto pořadí získáme překlad vstupního řetězce.

**Příklad 4.7.** Hrany v grafu závislosti na obr. 4.6 vycházejí vždy z uzlu s nižším číslem do uzlu s vyšším číslem. Topologický sort grafu závislosti tedy získáme zapsáním uzlů v pořadí

podle jejich čísel. Na základě topologického sortu pak můžeme zapsat následující program. (Pro atribut svázaný s uzlem $n$ grafu závislosti budeme používat označení $a_n$.)

$a_4 := real$;
$a_5 := a_4$;
$addtype(\mathbf{id}_3.entry, a_5)$;
$a_7 := a_5$;
$addtype(\mathbf{id}_2.entry, a_7)$;
$a_9 := a_7$;
$addtype(\mathbf{id}_1.entry, a_9)$;

Vyhodnocením těchto sémantických pravidel vložíme do tabulky symbolů pro všechny deklarované identifikátory typ *real*. ■

Pro vyhodnocování sémantických pravidel bylo navrženo několik metod.

- *Metody derivačního stromu.* Tyto metody získávají pořadí vyhodnocení sémantických pravidel v čase překladu z topologického sortu grafu závislosti, vytvořeného z derivačního stromu pro každý vstupní text. Pořadí vyhodnocení tyto metody nenajdou pouze v případě, že graf závislosti pro uvažovaný derivační strom obsahuje cyklus.
- *Metody založené na pravidlech.* V době vytváření překladače se analyzují sémantická pravidla spojená s pravidly gramatiky ručně nebo specializovanými prostředky. Pro každé pravidlo gramatiky je pořadí, ve kterém se budou vyhodnocovat příslušné atributy, pevně určeno již při návrhu překladače.
- *Nezávislé metody.* Pořadí vyhodnocení se vybere bez ohledu na sémantická pravidla. Například probíhá-li překlad během syntaktické analýzy, je pořadí vyhodnocení dáno použitou metodou překladu, nezávisle na sémantických pravidlech. Nezávislé pořadí vyhodnocování omezuje třídu syntaxí řízených definic, jež mohou být implementovány.

Metody založené na pravidlech a nezávislé metody nemusejí explicitně konstruovat během překladu graf závislosti, takže mohou být efektivnější s ohledem na dobu překladu i velikost požadované paměti.

## 4.3 Vyhodnocení S-atributových definic zdola nahoru

Vytvoření překladače pro obecnou syntaxí řízenou definici může být značně obtížný problém. Existují však dosti rozsáhlé třídy se speciálními vlastnostmi, pro které lze překladač implementovat jednoduše. Jednou z nich jsou S-atributové definice, tj. takové definice, které pracují pouze se syntetizovanými atributy.

Syntetizované atributy můžeme vyhodnocovat současně s analýzou zdrojového textu zdola nahoru. Atributy mohou být uloženy společně s ostatními informacemi, které používá analyzátor, na zásobníku. Při každé redukci podle nějakého pravidla se vypočtou atributy nonterminálního symbolu na levé straně pravidla a ty se uloží do zásobníku. Atributy symbolů na pravé straně jsou v okamžiku redukce umístěny na vrcholu zásobníku, takže jsou pro výpočet vždy k dispozici. Při vhodném návrhu překladového schematu je možné pracovat v omezené míře i s dědičnými atributy, jak dále ukážeme.

Syntaktický analyzátor pracující metodou zdola nahoru používá pro uchovávání informací o průběhu analýzy zásobník. Položky zásobníku můžeme rozšířit vždy o hodnotu atributu,



| | *state* | *val* |
|---|---|---|
| | . . . | . . . |
| | $X$ | $X.x$ |
| | $Y$ | $Y.y$ |
| *top* → | $Z$ | $Z.z$ |
| | . . . | . . . |

Obrázek 4.7: Rozšířený zásobník syntaktického analyzátoru

jak ukazuje obr. 4.7. Každá položka odpovídá vždy jednomu symbolu v již zpracované části větné formy; tento symbol je uveden ve sloupci *state*. Ve sloupci *val* je pak uvedena hodnota atributu odpovídajícího symbolu z prvního sloupce. Jinou možnou implementací je použití dvou paralelních zásobníků, jednoho pro uchovávání informací o analýze a druhého pro atributy. Pokud může mít jeden symbol více atributů, můžeme je všechny umístit do jednoho záznamu a tento nový datový typ pak používat jako jediný (strukturovaný) atribut. Rovněž mají-li různé symboly různé typy atributů, můžeme všechny tyto typy sloučit do jediného pomocí záznamu s variantními složkami, unie nebo podobné konstrukce, kterou pro to poskytuje implementační jazyk.

Současný vrchol zásobníku je označen ukazatelem *top*. Předpokládáme, že se syntetizované atributy vyhodnocují právě před provedením redukce. Máme-li například s pravidlem $A \to XYZ$ svázáno sémantické pravidlo $A.a := f(X.x, Y.y, Z.z)$, je před redukcí atribut $Z.z$ uložen ve $val[top]$, atribut $y.y$ ve $val[top-1]$ a atribut $X.x$ ve $val[top-2]$. Pokud symbol nemá atribut, není odpovídající hodnota pole *val* definována. Po redukci se hodnota *top* sníží o 2, stav odpovídající $A$ se uloží do $state[top]$ (tj. místo $X$) a vypočtená hodnota syntetizovaného atributu $A.a$ se uloží do $val[top]$.

**Příklad 4.8.** Uvažujme gramatiku z obr. 4.2 pro výpočet hodnoty aritmetického výrazu. Tato gramatika pracuje pouze se syntetizovanými atributy a může být tedy implementována přímo při překladu zdola nahoru. Opět předpokládáme, že lexikální analyzátor dodá hodnotu atributu $\mathbf{num}.ival$; tuto hodnotu uložíme do zásobníku při provádění akce přesun. Obr. 4.8 uvádí možnou implementaci sémantických akcí s atributy uloženými v poli *val*.

| PRAVIDLA | SÉMANTICKÁ AKCE |
|---|---|
| $E \to E_1 + T$ | $val[ntop] := val[top-2] + val[top]$ |
| $E \to T$ | |
| $T \to T_1 * F$ | $val[ntop] := val[top-2] * val[top]$ |
| $T \to F$ | |
| $F \to (\ E\ )$ | $val[ntop] := val[top-1]$ |
| $F \to \mathbf{num}$ | |

Obrázek 4.8: Implementace aritmetického výrazu pomocí atributového zásobníku

Uvedené úseky kódu pro sémantické akce neřeší nastavování proměnných *top* a *ntop*. Provádí-li se redukce podle pravidla s $r$ hodnotami na pravé straně, nastaví se *ntop* na

$top - r + 1$ a po provedení akce se *top* nastaví na *ntop*. Ještě vhodnější řešení je použít pro syntetizovaný atribut levé strany pravidla zvláštní proměnnou, která se pak přesune do zásobníku až po dokončení výpočtu. Obr. 4.9 ukazuje posloupnost akcí překladače při vstupu `2+3*5`.

| VSTUP | state | val | POUŽITÉ PRAVIDLO |
|---|---|---|---|
| 2+3*5 | - - | - - | |
| +3*5 | 2 | 2 | |
| +3*5 | $F$ | 2 | $F \to$ **num** |
| +3*5 | $T$ | 2 | $T \to F$ |
| +3*5 | $E$ | 2 | $E \to T$ |
| 3*5 | $E$ + | 3 - | |
| *5 | $E$ + 3 | 2 - 3 | |
| *5 | $E$ + $F$ | 2 - 3 | $F \to$ **num** |
| *5 | $E$ + $T$ | 2 - 3 | $T \to F$ |
| 5 | $E$ + $T$ * | 2 - 3 - | |
| | $E$ + $T$ * 5 | 2 - 3 - 5 | |
| | $E$ + $T$ * $F$ | 2 - 3 - 5 | $F \to$ **num** |
| | $E$ + $T$ | 2 - 15 | $T \to T * F$ |
| | $E$ | 17 | $E \to E + T$ |

Obrázek 4.9: Analýza a vyhodnocení výrazu `2+3*5`

Pro implementaci S-atributových definic je možné použít generátoru yacc. Implicitně yacc předpokládá, že všechny symboly jazyka mají jeden atribut typu `int`. Syntetizované atributy symbolů na pravé straně pravidla jsou dostupné pod symbolickým jménem `$`$i$, kde $i$ je pořadové číslo symbolu počínaje 1. Atribut levostranného neterminálu se ukládá do proměnné se symbolickým jménem `$$`. Nemá-li některé pravidlo uvedenu sémantickou akci, provede se implicitně akce { `$$ = $1;` }, která předá atribut prvního symbolu v pravidle jako atribut levé strany. Na obr. 4.10 je uvedeno totéž překladové schéma jako na obr. 4.8 zapsané pro generátor yacc.

```
%term NUM
%%
E  :  E '+' T   { $$ = $1 + $3; }
   |  T
   ;
T  :  T '*' F   { $$ = $1 * $3; }
   |  F
   ;
F  :  '(' E ')' { $$ = $2; }
   |  NUM
   ;
```

Obrázek 4.10: Specifikace překladu se syntetizovanými atributy pro yacc

Pokud potřebujeme jako atribut použít jiného datového typu, například v případě našeho aritmetického výrazu typ `double`, stačí do úvodní části specifikace doplnit text



```
%{
#define YYSTYPE double
%}
```

V praktických situacích však obvykle nevystačíme s jediným typem atributu pro všechny symboly. Jak již bylo uvedeno dříve, můžeme typ atributu definovat jako unii několika různých typů. K tomu nabízí yacc své vlastní prostředky, které mnohem zjednoduší zápis sémantických akcí. V definiční části specifikace můžeme uvést deklaraci všech složek unie, například

```
%union
{  int ival;
   double rval;
}
```

kterou definujeme celočíselný atribut *ival* a reálný atribut *rval*. Dále musíme uvést pro každý terminální a nonterminální symbol s atributem jméno jeho atributu (jméno odpovídající složky unie). Toto jméno pak bude yacc vždy automaticky přidávat ke všem odkazům na atributy příslušných symbolů a zároveň bude kontrolovat, zda jsou definovány typy atributů, na které se v sémantických pravidlech odkazujeme. Definice typu atributu pro terminální symboly se uvádí v lomených závorkách bezprostředně za klíčovým slovem **`%term`** a platí pro všechny terminální symboly definované za tímto klíčovým slovem. Pro nonterminální symboly se používá obdobná syntaxe s klíčovým slovem **`%type`**.

**Příklad 4.9.** Rozšíříme gramatiku z příkladu 4.8 o reálné konstanty s tím, že výpočet hodnoty výrazu se bude celý provádět v pohyblivé čárce. K tomu budeme potřebovat atribut *ival* typu `int` pro celočíselné konstanty (symbol `INUM`) a atribut *rval* typu `double` pro reálné konstanty a nonterminály. Výsledná specifikace pro yacc je uvedena na obr. 4.11. ■

```
%union { int ival; double rval; }
%term <ival> INUM
%term <rval> RNUM
%type <rval> E T F
%%
E  :  E '+' T   { $$ = $1 + $3; }
   |  T
   ;
T  :  T '*' F   { $$ = $1 * $3; }
   |  F
   ;
F  :  '(' E ')' { $$ = $2; }
   |  INUM
   |  RNUM
   ;
```

Obrázek 4.11: Specifikace překladu s atributy různých typů

## 4.4 L-atributové definice

Mnohem širší třídou syntaxí řízených definic jsou L-atributové definice, jejichž atributy se mohou vždy vypočítat během jednoho průchodu analyzátoru zdrojovým textem. Tato třída zahrnuje všechny syntaxí řízené definice založené na LL(1) gramatikách; po určitých úpravách je lze použít i při překladu zdola nahoru. Následující definice specifikuje vlastnosti L-atributových definic.

**Definice 4.6.** Syntaxí řízená definice je *L-atributová*, jestliže všechny dědičné atributy symbolů $X_j, 1 \leq j \leq n$ na pravé straně pravidla $A \rightarrow X_1 X_2 \cdots X_n$ závisejí pouze na

- atributech symbolů $X_1, X_2, \ldots, X_{j-1}$ vlevo od $X_j$ v témže pravidle a
- dědičných atributech symbolu $A$ na levé straně pravidla. ■

Poznamenejme, že každá S-atributová definice je zároveň L-atributová, neboť uvedená omezení se vztahují pouze na dědičné atributy.

Pro zápis L-atributových definic zavedeme pojem *překladové schéma* jako syntaxí řízenou definici, která umožňuje zápis sémantických akcí kdekoliv uvnitř pravé strany pravidla. Tyto sémantické akce budeme uzavírat do složených závorek a budeme předpokládat, že se provedou vždy před analýzou symbolů, které za nimi následují. Překladová schemata nám umožní definovat explicitně pořadí vyhodnocení sémantických akcí.

**Příklad 4.10.** Překlad výrazů s operátorem sčítání a celočíselnými konstantami můžeme popsat pomocí následujícího překladového schematu:

$$
\begin{array}{lcl}
E & \rightarrow & T\ R \\
R & \rightarrow & +\ T\ \{print('+')\}\ R_1 \mid \epsilon \\
T & \rightarrow & \mathbf{num}\ \{print(\mathbf{num}.val)\}
\end{array}
$$

■

Při návrhu překladových schémat musíme dbát na to, aby hodnota každého atributu byla dostupná v okamžiku, kdy se na ni odkazujeme. Obecně pokud máme dědičné i syntetizované atributy, je třeba dodržovat následující pravidla:

- Dědičný atribut symbolu na pravé straně pravidla se musí vypočítat akcí umístěnou před tímto symbolem.
- Akce se nesmí odkazovat na syntetizovaný atribut symbolu vpravo od ní.
- Syntetizovaný atribut symbolu na levé straně pravidla se může vypočítat až tehdy, jsou-li k dispozici hodnoty všech atributů, které používá. Výpočet tohoto atributu se obvykle umísťuje na konec pravé strany pravidla.

Následující překladové schéma nedodržuje první z uvedených tří podmínek:

$$
\begin{array}{lcl}
S & \rightarrow & A_1\ A_2\ \{A_1.in := 1; A_2.in := 2\} \\
A & \rightarrow & a\quad \{print(A.in)\}
\end{array}
$$

Dědičný atribut $A.in$ ve druhém pravidle totiž není v okamžiku pokusu o jeho tisk při analýze řetězce $aa$ definován, pokud procházíme derivačním stromem do hloubky. Průchod začne



uzlem $S$ a dále pokračuje v uzlech $A_1$ a $A_2$ ještě před tím, než se nastaví hodnoty $A_1.in$ a $A_2.in$. Umístíme-li akci definující hodnoty $A_1.in$ a $A_2.in$ mezi symboly $A$ na pravé straně pravidla $A \rightarrow A_1 A_2$, bude $A.in$ již v okamžiku tisku definováno.

V případě, že máme L-atributovou syntaxí řízenou definici, lze z ní vždy vytvořit překladové schéma, jež splňuje uvedené tři požadavky.

## 4.5 Překlad shora dolů

V této části si ukážeme, jak lze implementovat L-atributové definice během prediktivní analýzy. Budeme pracovat spíše s překladovými schematy než se syntaxí řízenými definicemi, neboť ta nám umožňují vyjádřit explicitně pořadí akcí a výpočtu atributů. Ukážeme si také, jak se dá odstranit levá rekurze z překladového schematu se syntetizovanými atributy.

### 4.5.1 Odstranění levé rekurze z překladového schematu

Mnoho aritmetických operátorů je asociativních zleva, takže je přirozené pro jejich syntaxi použít zleva rekurzivní gramatiky. Následující postup umožňuje odstranit levou rekurzi z překladového schematu se syntetizovanými atributy. Předpokládejme, že máme následující překladové schéma:

$$
\begin{array}{lcll}
A & \rightarrow & A_1\ Y & \{A.a := g(A_1.a, Y.y)\} \\
A & \rightarrow & X & \{A.a := f(X.x)\}
\end{array}
\tag{4.2}
$$

Všechny symboly mají syntetizované atributy pojmenované odpovídajícím písmenem malé abecedy, $f$ a $g$ jsou libovolné funkce.

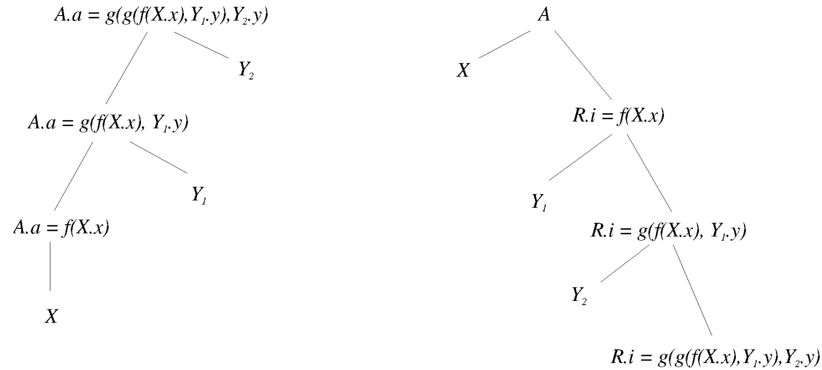
Algoritmem pro odstranění levé rekurze bychom z (4.2) dostali následující gramatiku:

$$
\begin{array}{lcl}
A & \rightarrow & X\ R \\
R & \rightarrow & Y\ R \mid \epsilon
\end{array}
\tag{4.3}
$$

Uvažujeme-li sémantické akce, získáme transformované schéma:

$$
\begin{array}{lcll}
A & \rightarrow & X & \{R.i := f(X.x)\} \\
 & & R & \{A.a := R.s\} \\
R & \rightarrow & Y & \{R_1.i := g(R.i, Y.y)\} \\
 & & R_1 & \{R.s := R_1.s\} \\
R & \rightarrow & \epsilon & \{R.s := R.i\}
\end{array}
\tag{4.4}
$$

Transformované schéma používá pro $R$ atributy $i$ a $s$. Aby bylo zřejmé, že výsledky (4.2) a (4.4) jsou shodné, uvažujme dva ohodnocené derivační stromy z obr. 4.12. Hodnota $A.a$ se na obr. 4.12(a) počítá podle (4.2). Obr. 4.12(b) obsahuje výpočet $R.i$ podle (4.4) při průchodu stromem směrem dolů. Hodnota $R.i$ se potom předává nahoru beze změny jako $R.s$ a nakonec se stane hodnotou atributu $A.a$ v kořeni ($R.s$ není na obr. 4.12(b) zakreslen).




Obrázek 4.12: Dva způsoby výpočtu atributů

### 4.5.2 Implementace prediktivního syntaxí řízeného překladače

L-atributové definice, jak již bylo uvedeno, umožňují vyhodnocení atributů v jediném průchodu již během syntaktické analýzy. Pro jejich implementaci můžeme použít metodu rekurzivního sestupu, která byla popsána v předchozí kapitole; rozšíříme ji pouze o sémantické akce a atributy. Atributový zásobník bude v tomto případě implementován podobně jako syntaktický zásobník pomocí implicitního zásobníku implementačního jazyka.

Atributy nonterminálních symbolů můžeme při rekurzivním sestupu reprezentovat jako parametry příslušných procedur. Dědičné atributy při tomto přístupu budou představovat vstupní parametry procedury (tj. parametry předávané hodnotou), syntetizované atributy naopak budou výstupními parametry (tj. budou předávány odkazem). V některých speciálních případech můžeme téhož parametru předávaného odkazem použít zároveň pro dva atributy — jeden dědičný a jeden syntetizovaný.

Atributy terminálních symbolů se vytvářejí v lexikálním analyzátoru a předávají se obvykle v globálních proměnných. Obsah příslušné globální proměnné můžeme podle potřeby uschovat pro pozdější použití do lokální proměnné definované uvnitř procedury.

Sémantické akce můžeme zapsat přímo na odpovídající místa v proceduře. Je však třeba dbát na to, aby se definovaly hodnoty všech syntetizovaných atributů levostranného nonterminálu (tj. hodnoty všech parametrů předávaných odkazem) i v případě, že dojde k syntaktické chybě, ze které se překladač zotaví. Pro tyto účely lze často použít speciálních hodnot atributů, které mohou být dále identifikovány a se kterými lze dále pracovat jako s neznámou informací.

**Příklad 4.11.** Uvažujme následující překladové schéma pro vyhodnocení výrazů s aditivními operátory a celočíselnými konstantami.

$$
\begin{aligned}
E &\rightarrow T\ \{R.i := T.val\ \}\ R\ \{E.val := R.s\} \\
R &\rightarrow \textbf{addop}\ T\ \{ \\
&\qquad\qquad \textbf{if addop}.op = add\ \textbf{then} \\
&\qquad\qquad\quad R_1.i := R.i + T.val \\
&\qquad\qquad \textbf{else} \\
&\qquad\qquad\quad R_1.i := R.i - T.val\} \\
&\qquad\quad R_1\ \{R.s := R_1.s\} \\
&\mid\ \epsilon\ \{R.s := R.i\} \\
T &\rightarrow \textbf{num}\ \{T.val := \textbf{num}.ival\}
\end{aligned}
$$



Symbol **addop** má atribut $op$ s hodnotou `'+'` nebo `'-'`; jeho hodnota bude uložena v globální proměnné $lexop$. Terminální symbol **num** představující celočíselnou konstantu má atribut $ival$, jehož hodnotu lexikální analyzátor uloží do globální proměnné $lexival$. Implementace tohoto překladového schematu je na obr. 4.13.

Nonterminál $R$ má dědičný atribut $i$, jehož hodnotou je vždy levý operand součtu nebo rozdílu, a syntetizovaný atribut $s$ představující mezivýsledek výpočtu hodnoty celého výrazu. Tyto dva atributy bychom mohli sloučit do jediného parametru procedury $R$ a tím celou implementaci zjednodušit. Povšimněte si, že některé sémantické akce, spočívající pouze v přiřazení hodnoty atributu, nejsou zapsány explicitně jako přiřazovací příkazy — například akce $\{R.i := T.val\}$ v pravidle pro nonterminál $E$ se realizuje předáním hodnoty proměnné `val1` jako argumentu procedury `R`. ■

## 4.6 Vyhodnocení dědičných atributů zdola nahoru

Pokud chceme implementovat L-atributovou definici během překladu zdola nahoru, narazíme na jeden zásadní rozdíl od přístupu shora dolů. Během analýzy zdola nahoru je pravidlo, podle kterého se bude redukovat, známo až v okamžiku redukce, tj. při dosažení jeho konce. To znamená, že všechny sémantické akce můžeme provádět až na konci pravidla. Přesto pomocí určitých transformací můžeme převést všechny L-atributové definice založené na LL(1) gramatikách do tvaru, který lze metodou zdola nahoru implementovat. Tyto transformace lze rovněž použít i na některé (ovšem ne všechny) definice založené na LR(1) gramatikách.

První užitečnou transformací je odstranění sémantických akcí, které jsou uvnitř pravidla. Tato transformace vkládá do původní gramatiky tzv. *marker*, nonterminální symbol generující prázdný řetězec $\epsilon$. Každou sémantickou akci, která je uvnitř pravé strany pravidla, nahradíme novým markerem $M$ a původní sémantickou akci přidáme na konec pravidla $M \rightarrow \epsilon$.

**Příklad 4.12.** Překladové schéma z příkladu 4.10 můžeme převést do tvaru

$$
\begin{aligned}
E &\rightarrow T\ R \\
R &\rightarrow +\ M\ R_1 \mid \epsilon \\
T &\rightarrow \textbf{num}\ \{print(\textbf{num}.val)\ \} \\
M &\rightarrow \epsilon\ \{print('+')\}
\end{aligned}
$$

který je ekvivalentní původnímu, tj. obě gramatiky přijímají stejný jazyk a pro všechny vstupní řetězce se sémantické akce provedou vždy ve stejném pořadí. Sémantické akce jsou nyní na koncích pravidel, takže je můžeme provést bezprostředně před redukcí. ■

```
procedure E(var val: integer);
var val1: integer;
begin
    T(val1);
    R(val1, val)
end;
procedure R(i: integer; var s: integer);
var op: char;
    val: integer;
begin
    if sym in ['+', '-'] then begin
        op := lexop;
        T(val1);
        if op = '+' then
            R(i + val1, s)
        else
            R(i - val1, s)
    end
    else
        s := i
end;
procedure T(var val: integer);
begin
    if sym = NUM then begin
        val := lexival;
        sym := lex
    end
    else
        error;
end;
```

Obrázek 4.13: Implementace překladového schematu rekurzivním sestupem

Pokud pracujeme s dědičnými atributy, můžeme využít toho, že během analýzy nonterminálu $Y$ v pravidle $A \rightarrow XY$ jsou na zásobníku stále k dispozici atributy symbolu $X$. Pokud je například dědičný atribut $Y.i$ symbolu $i$ definován pravidlem $Y.i := X.s$, kde $X.s$ je atribut symbolu $X$, můžeme místo hodnoty $Y.i$ všude použít $X.s$. Důležité je, aby tento atribut byl v zásobníku vždy na stejném místě.

**Příklad 4.13.** Uvažujme následující překladové schéma pro deklarace proměnných typu *integer* a *real*.



$$
\begin{array}{ll}
D \rightarrow T & \{\ L.in := T.type\ \} \\
\quad\ L & \\
T \rightarrow \textbf{int} & \{\ T.type := integer\ \} \\
T \rightarrow \textbf{real} & \{\ T.type := real\ \} \\
L \rightarrow & \{L_1.in := L.in\ \} \\
\quad\ L_1\ ,\ \textbf{id} & \{\ addtype(\textbf{id}.entry, L.in)\ \} \\
L \rightarrow \textbf{id} & \{\ addtype(\textbf{id}.entry, L.in)\ \}
\end{array}
$$

V okamžiku redukce libovolné pravé strany nonterminálu $L$ je na zásobníku symbol $T$ bezprostředně před touto pravou stranou. Místo atributu $L.in$, který je definován kopírovacím pravidlem $L.in := T.type$, tedy můžeme použít přímo atributu $T.type$. Uvedené schéma můžeme implementovat pomocí atributového zásobníku *val* tak, jak ukazuje obr. 4.14. Stejně jako na obr. 4.8 proměnná *top* obsahuje současný index vrcholu zásobníku a *ntop* index vrcholu zásobníku po provedení redukce. ■

| PRAVIDLO | SÉMANTICKÁ AKCE |
|---|---|
| $D \rightarrow T\ L\ ;$ | |
| $T \rightarrow \textbf{int}$ | $val[ntop] := integer$ |
| $T \rightarrow \textbf{real}$ | $val[ntop] := real$ |
| $L \rightarrow L\ ,\ \textbf{id}$ | $addtype(val[top], val[top-3])$ |
| $L \rightarrow \textbf{id}$ | $addtype(val[top], val[top-1])$ |

Obrázek 4.14: Implementace dědičných atributů při analýze zdola nahoru

Používáme-li pro generování syntaktického analyzátoru programu yacc, můžeme k atributům symbolů, které leží na zásobníku před pravou stranou redukovaného pravidla, přistupovat stejně jako k atributům symbolů redukovaného pravidla pomocí zápisu \$$i$, kde $i$ je index symbolu. Tento index je roven nule pro první symbol před redukovanou pravou stranou, -1 pro předcházející atd. Schéma z příkladu 4.13 tedy můžeme pro yacc zapsat tak, jak ukazuje obr. 4.15.

```
%term INT REAL ID
%%
D  :  T  L ;
T  :  INT  { $$ = integer; }
   |  REAL { $$ = real; } ;
L  :  L , ID  { addtype($3, $0); }
   |  ID      { addtype($1, $0); } ;
```

Obrázek 4.15: Použití dědičných atributů v zápisu pro yacc

V případě, že používáme atributy různých typů, nedovede yacc odvodit sám typ atributu, který nepatří symbolu v pravidle, a je tedy třeba tento typ uvést explicitně zápisem \$<*typ*>*i*. Yacc také umožňuje zápis sémantických akcí na libovolné místo pravé strany pravidla; případnou transformaci nahrazením sémantické akce markerem provede automaticky. Opět pokud má taková vnitřní akce syntetizovaný atribut a používáme-li typovaných atributů, je třeba uvést typ při všech odkazech na tento atribut.

# Kapitola 5

# Tabulka symbolů

V tabulce symbolů se uschovávají informace o pojmenovaných objektech, deklarovaných explicitně (uživatelské typy, proměnné, procedury, návěští atd.) nebo implicitně (standardní typy, procedury a funkce, pomocné proměnné vytvořené překladačem atd.). Tyto informace se využívají zejména k následujícím účelům:

- řešení kontextových vazeb v programu (vztah mezi deklarací a použitím objektu), které nelze popsat bezkontextovou gramatikou,
- provádění typové kontroly a
- generování intermediárního a cílového kódu.

Jednotlivé atributy objektů v tabulce symbolů jsou dány buď zdrojovým jazykem (např. jméno, druh, typ, počet parametrů procedury) nebo cílovým jazykem (např. velikost, adresa).

Tabulka symbolů se může vytvářet buď během sémantické analýzy a generování mezikódu — v tom případě předává lexikální analyzátor všechna jména jako řetězce znaků, — nebo se může vytvářet již během lexikální analýzy, kdy jsou jména objektů reprezentována v průběhu celého překladu pouze jako ukazatele do tabulky. Samozřejmě ve druhém případě musí sémantická analýza doplnit do tabulky zbývající údaje, které nemohou být po lexikální analýze ještě známé. Při jednoprůchodovém překladu může lexikální analyzátor přímo vyhledávat nalezené identifikátory v tabulce a umožnit syntaktické analýze využívat pro rozhodování některých kontextově závislých informací, např. místo symbolu pro identifikátor vrátit speciální symbol pro identifikátor proměnné nebo procedury. Taková interakce lexikálního analyzátoru s tabulkou symbolů může vést ke zjednodušení gramatiky a zlepšení detekce a zotavení se po kontextově závislých chybách, na druhé straně se ale snižuje modularita překladače.

## 5.1 Informace v tabulce symbolů

Kromě jmen objektů obsahuje tabulka symbolů — jak již bylo uvedeno na začátku této kapitoly — další informace potřebné pro činnost překladače. Strukturu těchto informací můžeme vyjádřit speciálním grafem, převzatým z teorie databázových systémů, tzv. *E–R grafem* (Entity–Relationship Graph — viz [6]). Tento graf vyjadřuje sémantické vztahy mezi jednotlivými objekty a dá se v překladači přímo implementovat pomocí dynamických datových struktur, jak si dále ukážeme. E–R graf má při překladu mnohem širší použití než jen pro popis informací v tabulce symbolů, ve skutečnosti umožňuje definovat úplný *sémantický model*



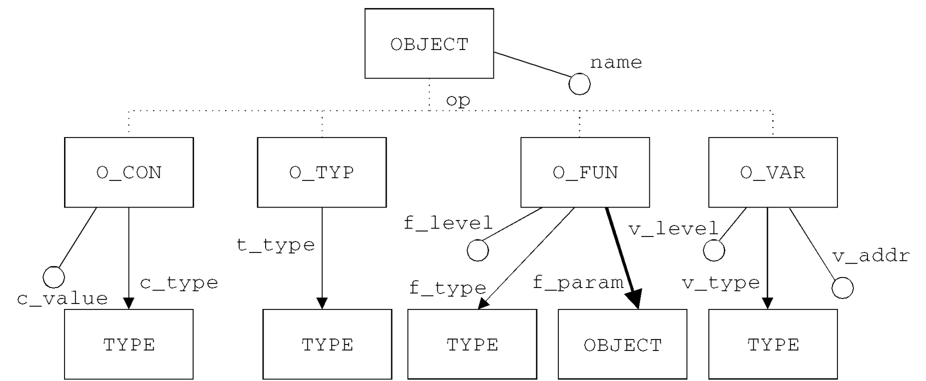


*programu*, kterým můžeme reprezentovat jak deklarace, tak i příkazy nebo výrazy v programu (viz článek 8.1.1).

E–R graf je tvořen dvěma množinami uzlů. Jedna množina uzlů představuje základní sémantické *entity* (pojmenované objekty, typy, příkazy, výrazy atd.) a druhá množina uzlů představuje *atributy* entit. Hrany spojující jednotlivé entity vyjadřují *relace* mezi entitami. Relace mohou být typu 1:1 (např. relace "typ proměnné" přiřazuje entitě "proměnná" její právě jeden typ) nebo typu 1:N (např. relace "parametr procedury" přiřazuje entitě "procedura" uspořádanou množinu objektů, reprezentujících její parametry). Atributy jsou spojeny hranami s uzly, k nimž patří (např. entita "proměnná" může mít jako atribut svou relativní adresu).

Graficky budeme entity znázorňovat obdélníky, relace 1:1 slabšími, relace 1:N silnějšími šipkami a atributy malými kroužky spojenými s entitami hranou. Jeden atribut může odlišovat různé varianty jediné entity (např. "objekt" může být "proměnná", "procedura", "návěští" atd.) — v tom případě tyto varianty nakreslíme jako samostatné entity spojené se společnou částí tečkovanými čarami.

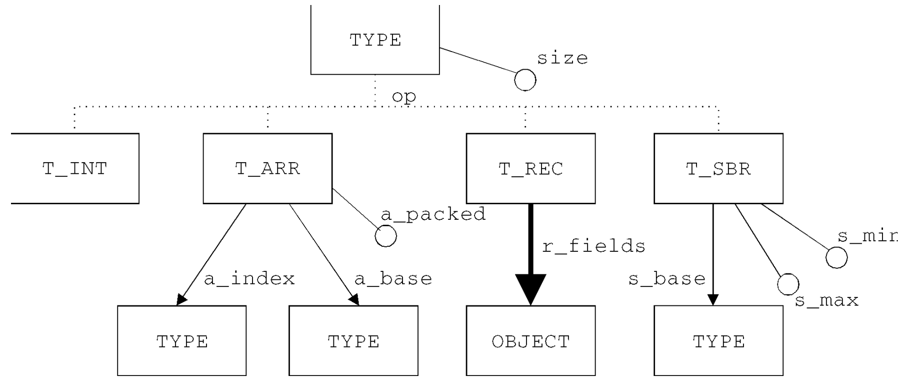
**Příklad 5.1.** Obr. 5.1 představuje část E–R grafu pro pojmenované objekty v jazyce Pascal. Tabulka symbolů bude v tomto případě uchovávat informace o entitách `OBJECT`, jejichž atribut `name` bude představovat vyhledávací klíč. Objekty mohou být konstanty, typy, funkce nebo proměnné, přičemž atribut `name` představuje jméno pojmenovaného objektu a atribut `op` rozlišuje druh objektu. Všechny uvedené objekty mají relací 1:1 definován typ, funkce má navíc seznam parametrů reprezentovaný relací `f_param` typu 1:N.




Obrázek 5.1: E–R graf pro objekty jazyka Pascal

Další entitou, která se na obr. 5.1 používá, je `TYPE`, reprezentující datový typ. Pro tuto entitu můžeme vytvořit stejným postupem graf, jehož část je na obr. 5.2. ■

Implementace E–R grafu pomocí dynamických datových struktur je již jednoduchá. Každou entitu budeme reprezentovat jedním záznamem, který bude obsahovat společné atributy a relace a případně seznam jednotlivých variant této entity. Vazby typu 1:1 můžeme definovat




Obrázek 5.2: E–R graf pro datové typy jazyka Pascal

jako ukazatele na příslušné typy entit, vazby 1:N jako ukazatele na první položku seznamu entit.

**Příklad 5.2.** Entitu `OBJECT` z příkladu 5.1 můžeme v Pascalu reprezentovat následujícími datovými typy:

```
type
   Objects = ( O_CON, O_TYP, O_FUN, O_VAR );
   ObjList = record
                ent:  ↑ObjEnt;
                next: ↑ObjList;
             end;
   ObjEnt  = record
                name: String;
                case op: Objects of
                   O_CON: ( c_value: Value;
                            c_type:  ↑TypeEnt );
                   O_TYP: ( t_type:  ↑TypeEnt );
                   O_FUN: ( f_level: Integer;
                            f_type:  ↑TypeEnt;
                            f_param: ↑ObjList );
                   O_VAR: ( v_level: Integer;
                            v_addr:  Integer;
                            v_type:  ↑TypeEnt );
                end;
   TypeEnt = ...
```

Při dalším zpracování takto reprezentovaného modelu deklarací je třeba mít stále na paměti, že výsledná datová struktura — i když se to tak jeví z uvedených příkladů — nemusí být



stromová. Například samotná reprezentace datového typu `ObjList` vede k cyklu v grafu (obsahuje ukazatel sama na sebe). Pro průchod sémantickým grafem se proto musejí využívat poněkud upravené algoritmy pro zpracování stromů. ■

## 5.2 Organizace tabulky symbolů

### 5.2.1 Operace nad tabulkou symbolů

Dvě nejběžněji prováděné operace nad tabulkou symbolů jsou operace *vkládání* (insertion) a *vyhledávání* (lookup, retrieval).

Operace vkládání do tabulky obecně nejprve zjistí, zda ukládaná hodnota klíče (v tomto případě objekt se stejným jménem) již v tabulce není. Pokud ne, vytvoří se nový záznam a zařadí se do tabulky. V opačném případě se může nahlásit chyba, např. “vícenásobně deklarovaný identifikátor.” U některých jazyků však nalezení jména v tabulce nemusí znamenat chybový stav, např. v následujících případech:

- deklarace procedury v Pascalu, jejíž záhlaví už bylo uvedeno dříve s direktivou *forward*,
- deklarace objektu, který byl už v programu použit, a kterému byly přiděleny implicitní atributy (např. funkce nebo návěští příkazu v jazyce C).

Operace vyhledání v tabulce obvykle vrátí informaci o tom, zda se objekt s požadovaným jménem v tabulce nachází, a v případě, že ano, vrátí rovněž nalezený objekt. Pokud objekt v tabulce není a zdrojový jazyk umožňuje implicitní deklarace, vytvoří se nový objekt s implicitními atributy, zařadí se do tabulky a vrátí se stejně, jako by v tabulce již byl.

V následujících odstavcích provedeme pouze přehled nejpoužívanějších metod. Implementace konkrétních algoritmů byla náplní kursu Programovací techniky (viz [5]).

### 5.2.2 Implementace tabulek pro jazyky bez blokové struktury

Pro jazyky bez blokové struktury vystačíme s jediným adresovým prostorem pro všechny položky. Některé z dále uvedených metod se rovněž používají pro vyhledávání v tabulce. Základní implementační metody jsou:

- *Neseřazené tabulky.* Neseřazené tabulky (pole, seznamy) jsou z hlediska implementace nejjednodušší. Položky do nich vkládáme v tom pořadí, jak jsou deklarované. Ukládání i vyhledávání má však časovou náročnost $\mathcal{O}(n)$, kde $n$ je počet položek v tabulce. Tato organizace se dá použít pouze tehdy, očekáváme-li malý počet položek.

- *Seřazené tabulky s binárním vyhledáváním.* Použijeme-li pro tabulku symbolů seřazené pole, můžeme snížit časovou náročnost vyhledávání na $\mathcal{O}(\log_2 n)$, ovšem nezmění se časová náročnost vkládání, neboť musíme stále zajišťovat seřazení tabulky. Binární vyhledávání v seřazeném poli je výhodné právě pro tabulky klíčových slov, které jsou statické.

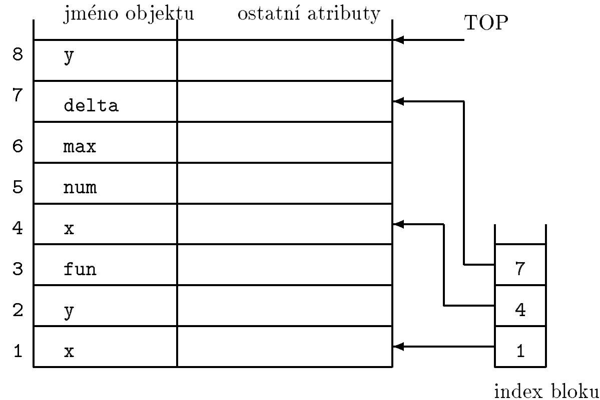
- *Stromově strukturované tabulky.* Stromové uspořádání tabulky symbolů redukuje dobu vkládání na $\mathcal{O}(\log_2 n)$. Doba vyhledávání se pohybuje mezi $\mathcal{O}(n)$ a $\mathcal{O}(\log_2 n)$, v závislosti na struktuře stromu. Tato doba je konstantní pro optimálně vyvážené stromy, které však vyžadují značně složité algoritmy vkládání. Proto se velmi často používají různá suboptimální řešení, nejčastěji *AVL stromy.*

- *Tabulky s rozptýlenými položkami.* Z hlediska doby vyhledávání jsou nejvýhodnějším řešením tabulky s rozptýlenými položkami, u nichž doba vyhledávání je do značné míry nezávislá na počtu záznamů v tabulce (závislost se projevuje až při vysokém zaplnění, kterému se dá předejít vhodnou volbou velikosti tabulky). Nevýhody této organizace jsou především v problematickém ošetření přeplnění tabulky, velkých nárocích na paměť a v tom, že tabulka neumožňuje systematický průchod položkami v abecedním pořadí.

### 5.2.3 Implementace blokově strukturované tabulky symbolů

Pro jazyky s blokovou strukturou jako Pascal, C nebo Modula-2 musí být k dispozici ještě další dvě operace, které označíme jako `tabopen` a `tabclose`. Operace `tabopen` se volá vždy na začátku nového bloku deklarací a operace `tabclose` na konci bloku. Tyto operace zajišťují rozlišování jednotlivých úrovní deklarací a umožňují uchovávat v tabulce několik různých objektů označených stejnými jmény za předpokladu, že byly deklarovány na různých úrovních. Operace vkládání a vyhledávání musejí proto splňovat ještě tyto dodatečné podmínky:

- při vkládání se pracuje pouze s naposledy otevřenou úrovní tabulky, případné další výskyty téhož jména na některé nižší úrovni se neberou v úvahu;
- při vyhledávání se prohledávají postupně všechny úrovně tabulky od nejvyšší úrovně k nejnižší a vrátí se objekt odpovídající prvnímu nalezenému výskytu hledaného jména.

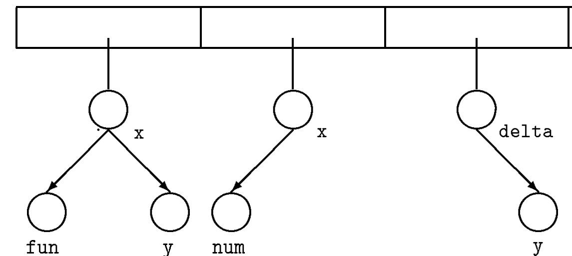



Obrázek 5.3: Příklad zásobníkové organizace tabulky symbolů s blokovou strukturou

Implementace blokově strukturované tabulky symbolů je obvykle založena na některé z metod, které byly uvedeny v předchozím odstavci. Vzhledem k tomu, že každá úroveň tabulky symbolů se uzavírá až tehdy, jsou-li uzavřené všechny vnořené úrovně, je přirozenou reprezentací blokově strukturované tabulky zásobník. V praxi se nejčastěji užívají tyto kombinace:

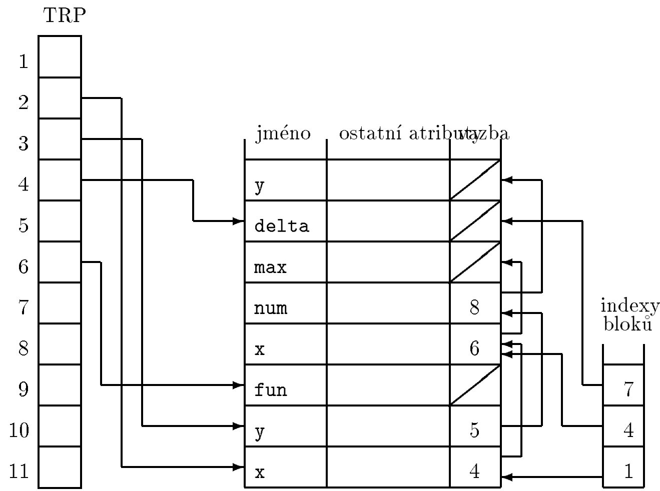


- *Zásobníková tabulka symbolů.* Jde o nejjednodušší organizaci tabulky, kdy jsou záznamy jednoduše umisťovány na vrchol zásobníku tak, jak přicházejí jednotlivé deklarace symbolů. Kromě zásobníku položek se ještě udržuje zásobník indexů úrovní, který ukazuje odkaz vždy na první položku dané úrovně (viz obr. 5.3). Operace `tabopen` pouze uloží na jeho vrchol současný index vrcholu zásobníku položek a operace `tabclose` vrátí index zásobníku položek na hodnotu, která leží na vrcholu zásobníku indexů. Při vyhledávání se prochází zásobník od vrcholu směrem zpět, při ukládání se hledají případné předchozí výskyty jména pouze na vrcholu zásobníku, až po naposledy uložený index. Tato organizace je velmi podobná nesetříděné tabulce symbolů včetně jejích nevhodných časových charakteristik, proto se dá použít pouze tam, kde se neočekává velký počet ukládaných položek.




Obrázek 5.4: Příklad stromově organizované tabulky symbolů s blokovou strukturou

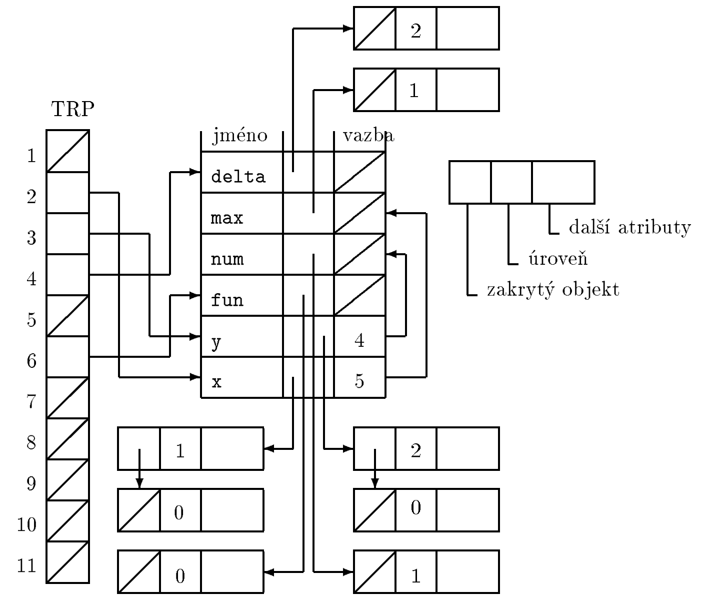
- *Kombinace zásobníku a stromu.* Při této organizaci udržujeme podobně jako v předchozím případě zásobník otevřených úrovní tabulky symbolů, ovšem tento zásobník nyní bude obsahovat odkazy na kořenové uzly stromů pro jednotlivé úrovně (viz obr. 5.4). Každé otevřené úrovni nyní přísluší samostatná tabulka symbolů, organizovaná jako strom. Při vkládání se pracuje pouze se stromem, na který ukazuje položka na vrcholu zásobníku úrovní, při vyhledávání se postupně procházejí jednotlivé úrovně počínaje naposledy otevřenou úrovní. Tato metoda je zvláště vhodná pro velký počet položek v tabulce, pokud jsme omezováni velikostí paměti.
- *Kombinace zásobníku a tabulky s rozptýlenými položkami.* Použití tabulky s rozptýlenými položkami pro blokově strukturované jazyky není příliš zřejmé, tento typ tabulky nezachovává pořadí položek a nemůže samostatně zajistit určitý způsob procházení tabulkou. Je však možné použít oddělený prostor pro položky a vlastní tabulku organizovat pouze jako tabulku ukazatelů na položky (viz obr. 5.5). V tom případě můžeme podobně jako v první uvedené metodě ukládat do zásobníku index první přidělené položky každé otevřené úrovně. Tím máme k dispozici informaci o příslušnosti položek tabulky do jednotlivých bloků, kterou můžeme využít při vyhledávání a vkládání. Operace `tabclose` kromě toho, že obnoví index vrcholu zásobníku položek, musí rovněž odstranit všechny příslušné odkazy a nahradit je příznaky neplatné položky (rušením položek v tabulce s rozptýlenými položkami se zabývá učební text [5]). Tato metoda vyžaduje při vkládání a vyhledávání projít celým řetězcem synonym a vyhledat v něm všechny výskyty téhož jména.




Obrázek 5.5: Příklad blokově strukturované tabulky s rozptýlenými položkami

- *Jednoúrovňová blokově strukturovaná tabulka symbolů.* Pravděpodobně nejefektivnější variantou blokově strukturované tabulky symbolů s rozptýlenými položkami využívá zásobníku pro ukládání všech existujících deklarací konkrétního identifikátoru (viz 5.6), zatímco hlavní vyhledávací mechanismus je implementován jedinou společnou vyhledávací tabulkou pro všechny úrovně. Je-li při operaci vkládání v tabulce nalezena položka se shodným jménem, avšak deklarovaná v nadřazené úrovni, přidá se do tabulky nová položka, na kterou se přesměruje původní odkaz, a do nové položky se uschová adresa zakryté položky. Tím se při vyhledávání zajistí, že budou přístupná pouze ta jména, která jsou zároveň dostupná na současné úrovni deklarací v programu. Operace `tabclose` musí v tabulce vyhledat všechny položky patřící do právě uzavírané úrovně, obnovit odkazy na zakrytá jména, případně zcela z tabulky odstranit odkazy na jména, která nebyla deklarována v žádném nadřazeném bloku. Podobná organizace se dá využít i pro implementaci tabulky pomocí binárních vyhledávacích stromů. Její hlavní výhoda je v tom, že doba vyhledávání není závislá na deklarační úrovni hledaného jména (vyhledávání probíhá paralelně na všech úrovních).






Obrázek 5.6: Příklad jednoúrovňové blokově strukturované tabulky symbolů

# Kapitola 6

# Struktura programu v době běhu

Ještě než začneme uvažovat generování kódu, musíme definovat vztah mezi statickým zdrojovým textem programu a akcemi, které se musejí provést v době běhu programu. Během zpracování může totéž jméno ve zdrojovém textu označovat různé objekty na cílovém počítači. Tato kapitola se bude zabývat vztahem mezi jmény a datovými objekty.

Přidělování a uvolňování paměti pro datové objekty má na starosti *systém řízení programu v době běhu* (run-time system), tvořený podprogramy zaváděnými společně s cílovým programem. Návrh řídicího systému je silně ovlivňován sémantikou procedur. V této kapitole se budeme zabývat technikami, které jsou využitelné pro jazyky jako je C, Pascal nebo Modula-2.

Každé provedení procedury nazýváme její *aktivací*. Je-li procedura rekurzivní, může v jednom okamžiku existovat zároveň několik jejích aktivací. Každé volání procedury v Pascalu vede k aktivaci, která může manipulovat s datovými objekty přidělenými speciálně pro její potřebu.

Reprezentace datových objektů v době běhu je dána jejich typem. Často lze elementární datové typy jako jsou znaky, celá a reálná čísla reprezentovat na cílovém počítači ekvivalentními datovými objekty. Složené datové typy jako jsou pole, řetězce a struktury, se obvykle reprezentují jako kolekce primitivních objektů.

## 6.1 Podprogramy

Většina současných procedurálních programovacích jazyků umožňuje vytváření strukturovaných programů, ve kterých je základním pojmem *podprogram* jako samostatná programová jednotka, představující abstrakci nějaké akce. Abychom byli konkrétní, budeme předpokládat, že zdrojový program je tvořen procedurami a funkcemi jako v Pascalu.

### 6.1.1 Statická a dynamická struktura podprogramů

*Definice podprogramu* je deklarace, která ve své nejjednodušší formě váže identifikátor s příkazem. Tento identifikátor je *jméno podprogramu* a příkaz je *tělo podprogramu*. Například úsek programu v Pascalu na obr. 6.1 obsahuje na řádcích 3–9 definici podprogramu se jménem `fib`; tělo podprogramu je na řádcích 5–8. Podprogramy, které vracejí hodnotu, se nazývají *funkce*, ostatní podprogramy se nazývají *procedury*. Celý program lze rovněž chápat jako podprogram volaný programy operačního systému počítače.





```
 (1)  program table(input,output);
 (2)     var max : integer;

 (3)     function fib(n: integer): integer;
 (4)        begin
 (5)           if n < 2 then
 (6)              fib := 1
 (7)           else
 (8)              fib := fib(n-2) + fib(n-1)
 (9)        end;

(10)     procedure printtab(n: integer);
(11)        var i : integer;
(12)        begin
(13)           for i := 1 to n do
(14)              writeln( i:3, fib(i):6 );
(15)        end;

(16)     begin
(17)        read(max);
(18)        printtab(max);
(19)     end.
```

Obrázek 6.1: Program v Pascalu pro tisk tabulky Fibonacciho čísel

Vyskytne-li se jméno podprogramu uvnitř proveditelného příkazu, říkáme, že se podprogram v tomto bodě *volá*. Volání podprogramu provede jeho tělo. Hlavní program na řádcích 16–19 v obr. 6.1 volá na řádku 18 proceduru `printtab`. Volání procedur má obvykle charakter příkazu, zatímco volání funkcí se vyskytuje jako součást výrazu.

Některé identifikátory v definici podprogramu jsou speciální a nazývají se *formální parametry* podprogramu. Identifikátor `n` je formálním parametrem procedury `fib`. Volanému podprogramu můžeme předat argumenty, nazývané také *skutečné parametry*; tyto argumenty nahrazují formální parametry podprogramu v jeho těle. Vztahem mezi skutečnými a formálními argumenty se budeme zabývat v článku 6.5. Na řádku 14 v obr. 6.1 je volání `fib` se skutečným parametrem `i`.

Každé provedení těla podprogramu nazýváme aktivací podprogramu. *Doba života* aktivace podprogramu `p` je posloupnost kroků mezi prvním a posledním krokem provádění těla podprogramu, včetně času stráveného prováděním podprogramů volaných z `p`, jimi volaných podprogramů atd.

Jsou-li $a$ a $b$ aktivace podprogramů, potom jejich doby života se buď nepřekrývají, nebo jsou do sebe zanořené. To znamená, že začne-li $b$ ještě před ukončením $a$, musí řízení opustit $b$ dříve než $a$. Tato vlastnost se dá využít při přidělování prostoru pro lokální proměnné podprogramů na zásobníku. Podprogram je *rekurzivní*, jestliže jeho nová aktivace může začít ještě předtím, než se ukončí jeho dřívější aktivace.

Podprogramy představují prostředek pro strukturalizaci programu. Na tuto strukturalizaci můžeme pohlížet ze dvou stran: jako na statické členění textu programu do samostatných jednotek, nebo jako na hierarchii aktivních podprogramů v době běhu programu.

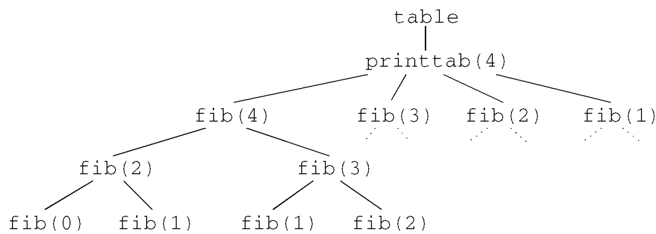
V jazycích jako je Pascal nebo Modula-2 mohou být uvnitř podprogramů deklarovány další podprogramy, které jsou v nich lokální. Každý podprogram má přiděleno číslo odpovídající jeho *statické úrovni zanoření*. Hlavní program má statickou úroveň 0, podprogramy v něm deklarované úroveň 1 atd. Například všechny funkce v jazyce C mají statickou úroveň 1.

Při běhu přeloženého programu dochází k volání jednotlivých podprogramů, které definuje implicitně *dynamickou úroveň zanoření*. Dynamickou strukturu programu můžeme znázornit *aktivačním stromem*, pro který platí následující pravidla:

1. každý uzel reprezentuje aktivaci podprogramu,
2. kořen reprezentuje aktivaci hlavního programu,
3. uzel $a$ je přímým předchůdcem uzlu $b$, právě když se řízení předává z aktivace $b$ do $a$,
4. uzel $a$ je uveden vlevo od uzlu $b$, právě když doba života $a$ předchází dobu života $b$.

Dynamická úroveň zanoření konkrétní aktivace podprogramu je potom rovna vzdálenosti příslušného uzlu od kořene aktivačního stromu.

**Příklad 6.1.** Aktivační strom na obr. 6.2 byl vytvořen pro program `table` z obr. 6.1 pro vstupní hodnotu `max` rovnou 4. Kořen stromu je tvořen hlavním programem, pod nímž následuje aktivace procedury `printtab` a dále jednotlivá rekurzivní volání funkce `fig`. ■




Obrázek 6.2: Aktivační strom

Při běhu programu má každá aktivace podprogramu obvykle k dispozici vlastní oblast paměti pro lokální proměnné a další pomocné údaje (obsah registrů v okamžiku volání, návratová adresa z podprogramu apod.). Tato oblast paměti se nazývá *aktivační záznam* podprogramu. Aktivační záznamy mohou mít v případě, že zdrojový jazyk neumožňuje rekurzivní volání, přidělenu statickou oblast paměti nebo se mohou uchovávat v zásobníku. Při volání podprogramu se na vrchol řídicího zásobníku uloží nový aktivační záznam, který se odstraní při návratu zpět. Je-li na vrcholu řídicího zásobníku aktivační záznam pro uzel $n$ aktivačního stromu, potom zbytek zásobníku obsahuje aktivační záznamy všech nadřazených uzlů v cestě od kořene k uzlu $n$. Blíže se budeme organizací paměti v době běhu zabývat v dalším článku.

## 6.2 Organizace paměti

Přeložený program dostane od operačního systému počítače k dispozici blok paměti, který obecně může být rozdělen na následující části:

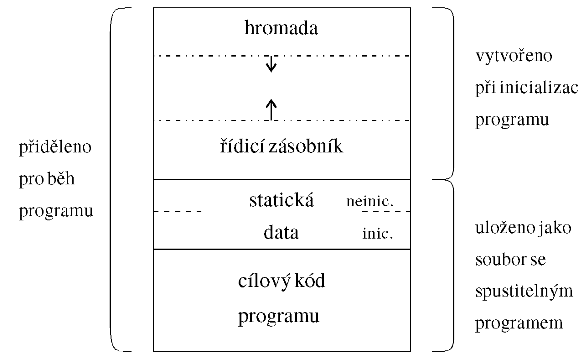


- vygenerovaný cílový kód,
- statická data,
- řídicí zásobník,
- hromada.

Velikost vygenerovaného kódu je známa již v době překladu, takže jej může překladač umístit do staticky definované oblasti, obvykle na začátek přiděleného paměťového prostoru. Rovněž velikost statických datových objektů může být známa již v době překladu a překladač je může umístit za program nebo uložit dokonce jako součást programu. Například v jazyce Fortran lze všem proměnným vyhradit prostor ve statické oblasti paměti, neboť neumožňuje rekurzivní volání podprogramů a pracuje pouze s daty, jejichž umístění lze definovat staticky v době překladu.

Jazyky umožňující rekurzivní volání procedur (C, Pascal) využívají pro aktivace podprogramů řídicího zásobníku, do kterého se ukládají jednotlivé aktivační záznamy. Strukturou aktivačního záznamu se budeme zabývat později.

Pro účely dynamického přidělování paměti (explicitně vyžádaného voláním příslušných funkcí nebo implicitně při přidělování paměti například pro pole s dynamickými rozměry) se používá zvláštní část paměti zvané *hromada*. Vzhledem k tomu, že se velikosti použité části paměti pro zásobník a hromadu v průběhu činnosti programu mohou značně měnit, je výhodné pro obě části využít opačné konce společné části paměti — viz obr. 6.3. Nedostatek paměti se rozpozná tehdy, jestliže ukazatel konce některé oblasti překročí hodnotu ukazatele konce druhé oblasti.




Obrázek 6.3: Organizace paměti při běhu programu

## 6.3 Strategie přidělování paměti

Pro datové oblasti, jimiž jsme se zabývali v předchozím článku, se používají následující hlavní metody přidělování paměti:

- statické přidělení paměti v době překladu,
- přidělování paměti na zásobníku a
- přidělování paměti z hromady.

V dalších odstavcích se zaměříme na přidělování paměti pro aktivační záznamy podprogramů.

### 6.3.1 Statické přidělování

Při statickém přidělování paměti jsou všem objektům v programu přiděleny adresy již v době překladu. Při kterémkoliv volání podprogramu jsou jeho lokální proměnné vždy na stejném místě, což umožňuje zachovávat hodnoty lokálních proměnných nezměněné mezi různými aktivacemi podprogramu. Statická alokace proměnných však klade na zdrojový jazyk určitá omezení. Údaje o velikosti a počtu všech datových objektů musejí být známy již v době překladu, rekurzivní podprogramy mají velmi omezené možnosti, neboť všechny aktivace podprogramu sdílejí tytéž proměnné, a konečně nelze vytvářet dynamické datové struktury.

Jedním z jazyků, které používají statické přidělování paměti, je Fortran. Program ve Fortranu se skládá z hlavního programu, podprogramů a funkcí. Aktivační záznamy podprogramů mohou být umístěny dokonce přímo v kódu, což se používalo běžně u starších počítačů.

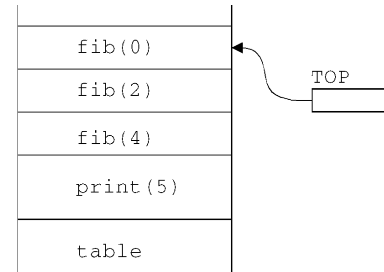
### 6.3.2 Přidělování na zásobníku

Přidělování paměti pro aktivační záznamy na zásobníku se používá běžně u jazyků, které umožňují rekurzivní volání podprogramů nebo které používají staticky do sebe zanořené podprogramy. Paměť pro lokální proměnné je přidělena při aktivaci podprogramu vždy na vrcholu zásobníku a při návratu je opět uvolněna. To ale zároveň znamená, že hodnoty lokálních proměnných se mezi dvěma aktivacemi podprogramu nezachovávají.

Při implementaci přidělování paměti na zásobníku bývá jeden registr vyhrazen jako ukazatel na začátek aktivačního záznamu na vrcholu zásobníku. Vzhledem k tomuto registru se pak počítají všechny adresy datových objektů, které jsou umístěny v aktivačním záznamu. Naplnění registru a přidělení nového aktivačního záznamu je součástí *volací posloupnosti*, obnovení stavu před voláním se provádí během *návratové posloupnosti*. Volací (a návratové) posloupnosti se od sebe v různých implementacích liší. Jejich činnost bývá rozdělena mezi volající a volaný program; obvykle volající program určí adresu začátku nového aktivačního záznamu (k tomu potřebuje znát velikost záznamu vlastního), přesune do něj předávané argumenty a spustí volaný podprogram zároveň s uložením návratové adresy do určitého registru nebo na známé místo v paměti. Volaný podprogram nejprve uschová do svého aktivačního záznamu stavovou informaci (obsahy registrů, stavové slovo procesoru, návratovou adresu), inicializuje svá lokální data a pokračuje zpracováním svého těla. Při návratu opět volaný podprogram uloží hodnotu výsledku do registru nebo do paměti, obnoví uschovanou stavovou informaci a provede návrat do volajícího programu. Ten si převezme návratovou hodnotu a tím je volání podprogramu ukončeno. Na obr. 6.4 je uveden stav řídicího zásobníku při vyhodnocování nejlevějšího koncového uzlu aktivačního stromu z obr. 6.2.

Umožňuje-li zdrojový jazyk předávat podprogramům datové struktury, jejichž velikost není známa v době překladu (např. pole, jehož počet prvků je dán hodnotou jiného parametru), je třeba uvedenou strategii poněkud modifikovat. V části aktivačního záznamu, kde






Obrázek 6.4: Řídicí zásobník

jsou umístěny parametry, se vyhradí pouze místo pro deskriptor objektu s ukazatelem na jeho skutečnou hodnotu a případně ještě dalšími informacemi, a pro vlastní objekt se vyhradí místo samostatně až za všemi položkami s pevnou délkou. K hodnotě objektu se pak přistupuje nepřímo přes deskriptor.

### 6.3.3 Přidělování z hromady

Strategie přidělování na zásobníku je nepoužitelná, pokud mohou hodnoty lokálních proměnných přetrvávat i po ukončení aktivace, případně pokud aktivace volaného podprogramu může přežít aktivaci volajícího. V těchto případech přidělování a uvolňování aktivačních záznamů se mohou překrývat, takže nemůžeme paměť organizovat jako zásobník.

Aktivační záznamy se mohou v těchto nejobecnějších situacích přidělovat z volné oblasti paměti (hromady), která se jinak používá pro dynamické datové struktury vytvářené uživatelem. Přidělené aktivační záznamy se uvolňují až tehdy, pokud se ukončí aktivace příslušného podprogramu nebo pokud už nejsou lokální data potřebná.

Při použití této strategie se pro vlastní přidělování a uvolňování paměti používají stejné techniky jako pro dynamické proměnné.

## 6.4 Metody přístupu k nelokálním objektům

V předchozích odstavcích jsme se zabývali různými metodami přidělování paměti pro lokální data podprogramů. Nebrali jsme však do úvahy existenci globálních dat — globálních datových objektů přístupných v rámci celého programu, případně lokálních proměnných ve staticky nadřazených podprogramech.

Data, která jsou globální v celém programu, mají charakter statických dat a může být pro ně použito techniky statického přidělování paměti. Adresy těchto objektů jsou známy již v době překladu. Například v jazyce C existují pouze globální data a lokální data jednotlivých funkcí, které do sebe nemohou být staticky zanořené.

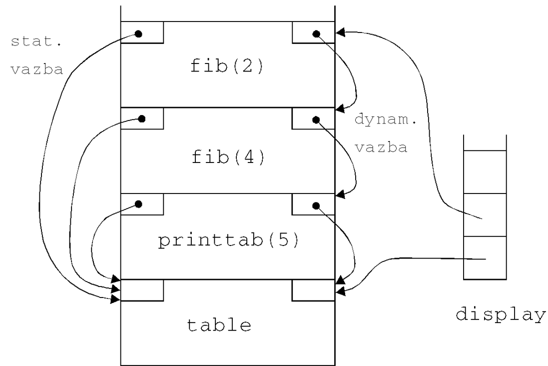
Pro podprogramy, které jsou staticky zanořené do jiných podprogramů, musíme zajistit možnost přístupu k lokálním proměnným nadřazených bloků, tj. k jejich aktivačním záznamům. Nejjednodušším řešením je rozšíření aktivačního záznamu o ukazatel na aktivační záznam bezprostředního staticky nadřazeného podprogramu (***přístupový ukazatel***). Odkazuje-li

se příkaz v proceduře `p` na statické úrovni $n_p$ na proměnnou `a` na statické úrovni $n_a$, se musí nejprve projít $n_p - n_a$ přístupovými ukazateli, čímž získáme adresu aktivačního záznamu obsahujícího proměnnou `a`. Tuto adresu pak můžeme již přímo použít pro zpřístupnění proměnné `a`, neboť její relativní adresa v aktivačním záznamu je známa.

Kód pro vytvoření přístupových ukazatelů je součástí volací posloupnosti podprogramu. Předpokládejme, že procedura `p` na statické úrovni $n_p$ volá proceduru `x` na statické úrovni $n_x$. Postup při vytváření přístupového ukazatele závisí na tom, zda je či není volaná procedura zanořená do volající.

1. Je-li $n_p < n_x$, je `x` zanořená mnohem hlouběji než `p` a musí tedy být deklarovaná uvnitř `p` (jinak by nebyla přístupná). Přístupový ukazatel volané procedury v tomto případě bude ukazovat na přístupový ukazatel volající procedury.
2. Je-li $n_p \geq n_x$, musí být nadřazené bloky jak volané, tak volající procedury na úrovních $1, 2, \ldots, n_x - 1$ stejné. Následuje-li volající procedura $n_p - n_x + 1$ přístupových ukazatelů, dostane se na nejvyšší úroveň, která staticky zahrnuje obě procedury, volající i volanou. Přístupový ukazatel volané procedury se pak nastaví tak, aby ukazoval na ukazatel nalezeného bloku.

Uvedená metoda zpřístupnění globálních objektů vyžaduje při každém přístupu ke globálnímu objektu generovat instrukce pro průchod přístupovými ukazateli. Tento proces se dá zrychlit, pokud udržujeme v paměti pole $d$ ukazatelů na aktivační záznamy, zvané *display*. Obsah tohoto pole je vždy takový, že hodnota $d[i]$ udává adresu aktivačního záznamu podprogramu na statické úrovni $i$ (viz obr. 6.5). Při volání podprogramu na statické úrovni $i$ nejprve musíme uschovat do nového aktivačního záznamu starou hodnotu $d[i]$ a potom nastavit $d[i]$ tak, aby ukazoval na nový aktivační záznam. Před ukončením aktivace pouze obnovíme uschovanou hodnotu $d[i]$.




Obrázek 6.5: Přístupové ukazatele a display

Display může být implementován různými způsoby. Pokud má cílový počítač dostatečný počet registrů, může být display tvořen posloupností vybraných registrů; tím se značně zjednoduší přístup k nelokálním proměnným, zvláště má-li cílový počítač instrukce s adresou danou součtem obsahu registru a nějaké konstanty. Překladač může na základě analýzy programu zjistit nejvyšší statickou úroveň zanoření, a tím i požadovaný počet registrů pro display, takže zbývající registry se mohou použít pro výpočty.



## 6.5 Předávání parametrů do podprogramů

Parametry podprogramu mají obvykle přidělen prostor v aktivačním záznamu. Do tohoto prostoru se při volání podprogramu umístí skutečné parametry — hodnoty, adresy, případně jiné datové struktury zpřístupňující předávaný parametr. To, co se konkrétně předává, závisí na typu a požadovaném způsobu předávání.

V této části se budeme zabývat několika technikami předávání parametrů. Na základě způsobu implementace můžeme tyto techniky rozdělit do tří skupin:

- *předávání hodnotou (kopírováním), výsledkem a hodnotou-výsledkem*
  Hodnota skutečného parametru se zkopíruje do formálního parametru nebo se výsledná hodnota formálního parametru zkopíruje zpět do skutečného parametru.
- *předávání odkazem (var)*
  Parametry předávané odkazem se reprezentují jako adresa skutečného parametru. Změna takového formálního parametru vede k bezprostřední změně skutečného parametru.
- *předávání jménem*
  Parametry předávané jménem se podle potřeby vyhodnocují při všech odkazech. Jejich zpracování je blízké zpracování makrodefinic.
- *předávání procedur a funkcí*
  Parametry, které představují procedury nebo funkce, se předávají jako deskriptory podprogramů; tyto deskriptory obsahují kromě adresy vstupního bodu podprogramu též vazbu reprezentující prostředí, v němž se má podprogram provádět.

### 6.5.1 Předávání parametrů hodnotou a výsledkem

Při předávání hodnotou se do aktivačního záznamu podprogramu zkopíruje hodnota skutečného parametru a veškeré výpočty uvnitř podprogramu se provádějí s touto kopií. To znamená, ze hodnota skutečného parametru se při tomto způsobu předávání nezmění. Parametry předávané hodnotou můžeme považovat za vstupní parametry podprogramu. Podobně při předávání výsledkem se v podprogramu pracuje stále s lokální hodnotou formálního parametru, která se při návratu z podprogramu okopíruje do skutečného parametru (skutečným parametrem tedy musí být L-hodnota, tj. taková hodnota, která může stát na levé straně přiřazení). Parametry předávané výsledkem mohou být pouze výstupními parametry. Kombinací obou metod získáme zároveň vstupní i výstupní parametr.

Tento způsob předávání parametrů můžeme implementovat jednoduše v místě volání, kdy přesuneme hodnotu parametru do nebo z aktivačního záznamu volaného podprogramu. Uvnitř podprogramu s takovým parametrem zacházíme stejně jako s kteroukoliv jinou lokální proměnnou. Poněkud odlišný přístup je třeba volit při předávání polí nebo řetězců. Zde se často využívá nepřímého přístupu přes přístupový vektor (deskriptor), který obsahuje adresu

začátku pole nebo řetězce a případně i další údaje, jako počet prvků pole, délku řetězce nebo rozsahy indexů. Takto je možné implementovat i předávání polí a řetězců proměnné délky. Velikost přístupového vektoru je známa v době překladu a je tedy možné pro něj vyhradit pevné místo v aktivačním záznamu. Skutečná hodnota pak může být uložena na jiném místě, např. v oblasti pro dynamické proměnné. Při předávání záznamů můžeme přesunout přímo hodnotu záznamu nebo předat jen jeho adresu a nechat vlastní přesun na volaném podprogramu.

### 6.5.2 Předávání parametrů odkazem

Při této metodě předávání parametrů umístí volající do aktivačního záznamu volaného podprogramu pouze adresu předávané l-hodnoty. Uvnitř podprogramu se pak všechny odkazy na takový formální parametr zpracovávají jako nepřímé. Pro pole můžeme předat přímo adresu začátku pole nebo adresu přístupového vektoru. Předávání parametrů odkazem se dá jednoduše nahradit předáváním adres parametrů hodnotou, například jako je to definováno v jazyce C. Pokud však takový jazyk nemá dostatečně silnou typovou kontrolu, může velmi často docházet k chybám, například pokud programátor předá místo ukazatele přímo hodnotu nebo naopak pokud místo hodnoty formálního parametru pracuje s jeho adresou.

**Příklad 6.2.** Následující podprogram v jazyce C provádí záměnu hodnot dvou proměnných, jejichž adresy jsou předávány hodnotou. Všechny výskyty parametrů ve výrazech musejí explicitně obsahovat dereferenci ukazatele.

```
void swap(int *x, int *y)
{
    int temp;
    temp = *x; *x = *y; *y = temp;
}
```

■

### 6.5.3 Předávání parametrů jménem

Metoda předávání parametrů jménem byla použita například v jazyce Algol 60. Je-li jako skutečný parametr předán výraz, např. odkaz na prvek pole `a[i]`, závisí v každém okamžiku jeho hodnota nejen na obsahu pole `a`, ale i na hodnotě proměnné `i`. Každý výskyt formálního parametru předávaného hodnotou v textu podprogramu se vlastně nahradí textově hodnotou skutečného parametru, jako by šlo o makrodefinici.

**Příklad 6.3.** Volání `swap(i, a[i])` podprogramu z příkladu 6.2 by se provedlo tak, jako bychom zapsali

```
temp := i; i := a[i]; a[i] := temp
```

To znamená, že při volání jménem se sice `i` nastaví na `a[i]` tak, jak očekáváme, avšak počáteční hodnotu $I_0$ proměnné `i` uloží do `a[a[`$I_0$`]]` a ne do `a[`$I_0$`]`. Lze ukázat, že pokud se používá předávání jménem, nelze správně pracující verzi procedury `swap` vůbec napsat. ■

Implementace předávání parametru jménem je značně obtížná. Pro každý takový parametr musíme vygenerovat podprogram pro jeho vyhodnocení. Další komplikací je, že vyhodnocení parametru musí probíhat v prostředí volajícího podprogramu (například pro odkazy na proměnné se musí použít tabulka symbolů platná v místě volání). Podprogramu se tedy předává



dvojice hodnot — adresa podprogramu pro vyhodnocení parametru a adresa definující prostředí v místě volání. Vzhledem k problematické implementaci se dnes metoda předávání parametrů jménem nepoužívá, je však zajímavá z hlediska vývoje jazyků a implementačních technik. Tato metoda je také velice blízká technice tzv. *otevřených* (inline) *podprogramů*, tj. podprogramů, jejichž tělo se vždy rozvine v místě volání.

### 6.5.4 Předávání procedur a funkcí

Při předávání podprogramu jako parametru musíme v jazycích, které umožnují zanořování podprogramů, řešit obdobný problém jako při předávání parametrů jménem. Nestačí pouze předat adresu začátku podprogramu — předávaný podprogram musí mít v okamžiku volání připraveno totéž prostředí, jako by byl volaný v místě předávání. Jedná se především o vazby zajišťující přístup ke staticky nadřazeným lokálním proměnným.

```
procedure A;
   var m: real;

   procedure B(procedure P);
   begin
      P
   end;

   procedure C;
      var x: real;
      procedure D;
      begin
         x := 3.25;
      end;
      procedure E;
      begin
         B(D)
      end;
   begin
      E
   end;
begin
   C
end;
```

Obrázek 6.6: Předávání procedury `D` jako parametru

Například v programu na obr. 6.6 procedura `E` volá proceduru `B` a předává jí jako parametr proceduru `D`. Procedura `D` musí mít přístupné proměnné `m` a `x`, avšak v místě jejího volání (v těle procedury `B`) je přístupná pouze proměnná `m`. Proto musí překladač zajistit kromě předání adresy `D` také předání ukazatele na aktivační záznam procedury `C` a při volání formální procedury zajistit potřebné vazby.

# Kapitola 7

# Typová kontrola

Překladač musí kontrolovat, zda zdrojový program dodržuje jak syntaktické, tak sémantické konvence zdrojového jazyka. Tato kontrola, zvaná statická kontrola (pro odlišení od dynamické kontroly během provádění cílového programu), zajišťuje detekci a ohlášení určitých druhů programátorských chyb. Příklady statických kontrol mohou být:

- *Typová kontrola.* Překladač by měl ohlásit chybu, pokud se nějaký operátor aplikuje na nekompatibilní operandy; například tehdy, jestliže se sečítá proměnná typu pole s proměnnou typu funkce.

- *Kontrola toku řízení.* Příkazy, které způsobí, že tok řízení opustí určitou konstrukci, musí mít určité místo, na které se má řízení přenést. Například příkaz `break` v C způsobí, že tok řízení opustí nejmenší obklopující příkaz `while`, `for` nebo `switch`; chyba nastane, pokud takový obklopující příkaz neexistuje.

- *Kontrola jedinečnosti.* Mohou nastat situace, kdy určitý objekt musí být deklarován právě jednou. Například v Pascalu musí být identifikátor deklarován jedinečně, návěští v příkazu case musejí být navzájem různá a prvky výčtového typu se nemohou opakovat.

- *Kontroly vztahující se ke jménům.* Někdy se určité jméno musí vyskytnout dvakrát nebo vícekrát. Například v jazyku Modula-2 musí být jméno procedury uvedeno znovu na jejím konci. Překladač musí zkontrolovat, zda je na obou místech použito totéž jméno.

V této kapitole se zaměříme na typovou kontrolu. Jak naznačují uvedené příklady, mnoho statických kontrol je rutinních a mohou se implementovat metodami z předchozí kapitoly. Některé z nich lze zahrnout do jiných činností. Například při vkládání informací do tabulky symbolů můžeme zkontrolovat, zda je jméno deklarováno jedinečně. Mnoho překladačů Pascalu kombinuje statickou kontrolu a generování intermediárního kódu se syntaktickou analýzou. Pro složitější konstrukce, jako jsou např. v jazyku Ada, může být vhodnější mít oddělený průchod provádějící typové kontroly mezi syntaktickou analýzou a generováním intermediárního kódu.

Podsystém typové kontroly ověřuje, zda typy konstrukcí odpovídají typům očekávaným z jejich kontextu. Například standardní aritmetický operátor `mod` jazyka Pascal vyžaduje celočíselné operandy, takže typová kontrola musí ověřit, zda oba operandy `mod` mají typ integer. Podobně musí typová kontrola prověřit, zda je operátor dereference aplikován na





ukazatel, že indexování se provádí pouze pro pole, že uživatelem definovaná funkce se aplikuje na správný počet a typ argumentů atd.

Informace o typech, získaná během typové kontroly, může být požadována při generování kódu. Například aritmetické operátory jako je `+` se obvykle aplikují buď na celá nebo na reálná čísla, a musíme tedy na základě kontextu rozhodnout, o který význam operátoru `+` se jedná. Symbol reprezentující v různých kontextech různé operace se nazývá *přetížený.* Přetěžování může být doprovázeno implicitní konverzí typů, kdy překladač doplňuje operátor pro konverzi operandu na typ očekávaný podle kontextu.

Odlišným pojmem od přetěžování je *polymorfismus.* Polymorfické funkce a procedury mohou při každém volání pracovat s argumenty jiných typů. Např. v jazyce Pascal můžeme proceduru `writeln` považovat za polymorfickou, neboť jejími argumenty mohou být celočíselné, reálné, booleovské výrazy, znaky nebo řetězce. V závislosti na typu skutečného argumentu se teprve vybírá konkrétní algoritmus pro zobrazení hodnoty.

## 7.1 Typové systémy

Návrh podsystému typové kontroly jazyka je založen na informacích o syntaktických konstrukcích jazyka a pravidlech pro přiřazování typů jazykovým konstrukcím. Tato pravidla mohou mít například následující formu:

- "Jsou-li oba operandy aritmetických operací sčítání, odčítání a násobení typu integer, je výsledek typu integer."

- "Výsledek unárního operátoru & je ukazatel na objekt, ke kterému se vztahuje operand. Je-li typ operandu '...', je typ výsledku 'ukazatel na ...'."

V uvedených úsecích se implicitně předpokládá, že s každým výrazem je svázán jeho typ. Typy navíc mohou mít určitou strukturu; typ "ukazatel na ..." je vytvořen z typu "...", na který se odkazuje.

V běžných programovacích jazycích jsou k dispozici obvykle dvě skupiny datových typů: základní nebo složené. Základní typy jsou atomické typy, z hlediska programátora bez další vnitřní struktury. V Pascalu jsou například základními typy *boolean*, *char*, *integer* a *real*. Intervaly jako 1..10 a výčtové typy jako

```
(violet, indigo, blue, green, yellow, orange, red)
```

lze považovat za základní typy. Pascal programátorovi dovoluje vytvářet podle potřeby další typy ze základních a dříve definovaných složených typů; příkladem jsou pole, záznamy a množiny. Jako složené typy lze navíc chápat i ukazatele a funkce.

### 7.1.1 Typové výrazy

Typ jazykové konstrukce lze popsat *typovým výrazem.* Neformálně je typový výraz buď základní typ nebo je vytvořen aplikací operátoru zvaného konstruktor typu na jiné typové výrazy. Soubor základních typů a konstruktorů je dán definicí jazyka.

V této kapitole budeme používat následující definice typového výrazu:

1. Základní typ je typový výraz. Mezi základními typy jsou *boolean*, *char*, *integer* a *real*. Speciální základní typ *type_error* signalizuje chybu během typové kontroly. Konečně

základní typ *void* označuje "nepřítomnost hodnoty" a dovoluje přiřadit datový typ i procedurám a příkazům.

2. Vzhledem k tomu, že typové výrazy mohou být pojmenované, je jméno typu typovým výrazem. Příklad použití jmen typů je dále v 3(c).

3. Typový konstruktor aplikovaný na typový výraz je typovým výrazem. Mezi konstruktory patří:

   a) *Konstruktor pole.* Je-li $T$ typový výraz, pak $array(I, T)$ je typovým výrazem, jenž označuje pole prvků typu $T$ s indexovou množinou $I$. Typ $I$ je často intervalem celých čísel. Například deklarace v Pascalu

```
var A: array [1..10] of integer;
```

   spojuje se jménem `A` typový výraz $array(1..10, integer)$.

   b) *Součin typů.* Jsou-li $T_1$ a $T_2$ typové výrazy, potom jejich kartézský součin $T_1 \times T_2$ je typovým výrazem. Předpokládáme, že $\times$ je zleva asociativní.

   c) *Záznamy.* Rozdíl mezi záznamem a součinem je ten, že složky záznamu jsou pojmenované. Typový konstruktor *record* bude aplikován na $n$-tici tvořenou jmény složek a typy složek. Například úsek programu v Pascalu:

```
type row = record
               address: integer;
               lexeme: array [1..15] of char
           end;
var table: array [1..101] of row;
```

   deklaruje jméno typu `row` představujícího typový výraz

$$record((\texttt{address} \times integer) \times (\texttt{lexeme} \times array(1..15, char)))$$

   a proměnnou `table` jako pole záznamů tohoto typu.

   d) *Ukazatele.* Je-li $T$ typový výraz, potom $pointer(T)$ je typový výraz označující typ "ukazatel na objekt typu $T$." Například opět v Pascalu deklarace

```
var p: ↑row
```

   deklaruje proměnnou `p` s typem $pointer(row)$.

   e) *Funkce.* Z matematického hlediska funkce zobrazuje prvky jedné množiny, definičního oboru, do jiné množiny, oboru hodnot. Funkce v programovacích jazycích můžeme chápat jako zobrazení zdrojového typu $D$ (domain) do cílového typu $R$ (range). Typ takové funkce budeme zapisovat typovým výrazem $D \to R$. Například standardní funkce `mod` jazyka Pascal má zdrojový typ $int \times int$, tj. dvojici celých čísel, a cílový typ $int$. Za předpokladu, že $\times$ má vyšší prioritu než $\to$ a že $\to$ je asociativní zprava, tedy má `mod` typ

$$int \times int \to int$$

   Jako další příklad vezmeme deklaraci z Pascalu



```
function f(a, b: char): ↑integer; ...
```

   která říká, že zdrojovým typem funkce `f` je $char \times char$ a cílovým typem je $pointer(integer)$. Typ `f` je tedy označen typovým výrazem

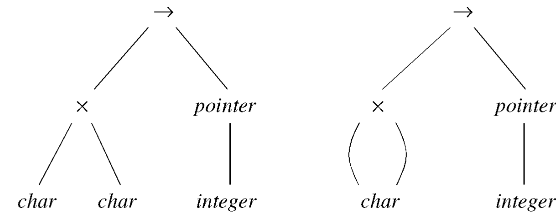
$$char \times char \to pointer(integer)$$

   Z implementačních důvodů jsou často kladena omezení na typ, jenž může funkce vracet; např. v jazyce C nelze vracet pole nebo funkce. Existují však jazyky, z nichž Lisp je nejvýraznějším příkladem, které dovolují, aby funkce vracely objekty libovolných typů, takže můžeme např. definovat funkci `g` typu

$$(integer \to integer) \to (integer \to integer),$$

   Funkce `g` tedy má jako argument funkci zobrazující celé číslo na celé číslo, a tato funkce produkuje jako výsledek jinou funkci stejného typu. Zpracování takovýchto funkcí (tzv. *funkcí vyššího řádu*) je typické pro funkcionální jazyky.

Výhodnou metodou reprezentace typových výrazů je použití grafu. Během překladu definice typu můžeme pro typový výraz sestrojit strom nebo DAG, jehož vnitřními uzly budou konstruktory typu a listy budou základními typy, jména typů a typových proměnných (viz obr. 7.1). Obdobnou reprezentací je grafový model, uvedený na obr. 5.2.




Obrázek 7.1: Strom a DAG pro výraz $char \times char \to pointer(integer)$

Typový systém je soubor pravidel pro přiřazování typových výrazů různým částem programu; v této kapitole jej budeme implementovat pomocí syntaxí řízeného překladu. Různými překladači téhož jazyka mohou být implementovány různé typové systémy. Například v systému Unix jsou pro původní verzi jazyka C k dispozici dva programy s odlišnými typovými systémy. Program `lint` provádí pouze statickou kontrolu programu bez jeho překladu, ovšem na základě mnohem přísnějšího typového systému než překladač `cc`, a tím umožňuje odhalení programátorských chyb, které samy o sobě nejsou v rozporu s definicí jazyka C.

**Příklad 7.1.** Jako příklad implementace typové kontroly použijeme jednoduchý jazyk, ve kterém musí být typ každého identifikátoru deklarován před jeho použitím. Jazyk má následující gramatiku:

$$\begin{array}{lcl} P & \to & D \; ; \; E \\ D & \to & D \; ; \; D \mid \textbf{id} \; : \; T \end{array}$$

$$T \;\rightarrow\; \textbf{char} \mid \textbf{integer} \mid \textbf{array}\ [\ \textbf{num}\ ]\ \textbf{of}\ T \mid \uparrow T$$
$$E \;\rightarrow\; \textbf{literal} \mid \textbf{num} \mid \textbf{id} \mid E\ \textbf{mod}\ E \mid E\ [\ E\ ] \mid E\hat{}$$

Základními typy jazyka jsou *char* a *integer*, typ *type_error* se používá pouze pro signalizaci typové chyby. Pro jednoduchost předpokládáme, že index pole začíná vždy od hodnoty 1. Překladové schéma na obr. 7.2 popisuje budování typových výrazů, deklaraci proměnných a typovou kontrolu výrazů. Po vhodné modifikaci gramatiky můžeme toto schéma použít jak pro překlad shora dolů, tak i pro překlad zdola nahoru.

| | |
|---|---|
| $P \rightarrow D\ ;\ E$ | |
| $D \rightarrow D\ \ D$ | |
| $D \rightarrow \textbf{id}\ :\ T$ | $\{\ \mathit{addtype}(\textbf{id}.\mathit{entry}, T.\mathit{type})\ \}$ |
| $T \rightarrow \textbf{char}$ | $\{\ T.\mathit{type} := \mathit{char}\ \}$ |
| $T \rightarrow \textbf{integer}$ | $\{\ T.\mathit{type} := \mathit{integer}\ \}$ |
| $T \rightarrow \hat{}\,T_1$ | $\{\ T.\mathit{type} := \mathit{pointer}(T_1.\mathit{type})\ \}$ |
| $T \rightarrow \textbf{array}\ [\ \textbf{num}\ ]\ \textbf{of}\ T_1$ | $\{\ T.\mathit{type} := \mathit{array}(1..\textbf{num}.\mathit{val}, T_1.\mathit{type})\ \}$ |
| $E \rightarrow \textbf{literal}$ | $\{\ E.\mathit{type} := \mathit{char}\ \}$ |
| $E \rightarrow \textbf{num}$ | $\{\ E.\mathit{type} := \mathit{integer}\ \}$ |
| $E \rightarrow \textbf{id}$ | $\{\ E.\mathit{type} := \mathit{lookup}(\textbf{id}.\mathit{entry})\ \}$ |
| $E \rightarrow E_1\ \textbf{mod}\ E_2$ | $\{\ E.\mathit{type} :=$ **if** $E_1.\mathit{type} = \mathit{integer}$ **and** $E_2.\mathit{type} = \mathit{integer}$ **then** *integer* **else** *type_error* $\}$ |
| $E \rightarrow E_1\ [\ E_2\ ]$ | $\{\ E.\mathit{type} :=$ **if** $E_2.\mathit{type} = \mathit{integer}$ **and** $E_1.\mathit{type} = \mathit{array}(s,t)$ **then** $t$ **else** *type_error* $\}$ |
| $E \rightarrow E_1\ \hat{}$ | $\{\ E.\mathit{type} :=$ **if** $E_1.\mathit{type} = \mathit{pointer}(t)$ **then** $t$ **else** *type_error* $\}$ |

Obrázek 7.2: Překladové schéma pro typovou kontrolu deklarací a výrazů

V uvedeném překladovém schematu akce $\mathit{addtype}(\textbf{id}.\mathit{entry}, T.\mathit{type})$ do položky tabulky symbolů specifikované syntetizovaným atributem *entry* uloží typ identifikátoru **id** z deklarace. Syntetizovaný atribut *type* nonterminálu $E$ udává typ odpovídajícího výrazu. Pro zjištění typu, který je svázán s položkou tabulky symbolů $e$, používáme funkce $\mathit{lookup}(e)$

Při kontrole operátoru **mod** ve výrazu požadujeme, aby oba operandy měly typ *integer*. V odkazu na prvek pole $E_1[E_2]$ musí mít indexový výraz $E_2$ typ *integer*; typ výsledku $t$ je potom dán typem prvku pole, který získáme z konstruktoru $\mathit{array}(s,t)$. Pro výraz $E\hat{}$ požadujeme, aby jeho operandem byl ukazatel; typ $t$ celého výrazu opět získáme z konstruktoru $\mathit{pointer}(t)$. Toto překladové schéma můžeme podobným způsobem rozšířit o další typy a operátory.



### 7.1.2 Statická a dynamická kontrola typů

Kontrole prováděné překladačem říkáme statická, zatímco kontroly prováděné při běhu programu se nazývají dynamické. V principu je možné všechny kontroly provádět až dynamicky, pokud cílový kód ponese s hodnotou prvku zároveň i jeho typ. Z hlediska efektivity spolehlivosti programu je však vhodnější provádět v době překladu co největší počet kontrol.

*Spolehlivý typový systém* (sound type system) vylučuje potřebu dynamické kontroly typových chyb, neboť dovoluje staticky zajistit, že takové chyby nemohou za běhu cílového programu nastat. To znamená, že pokud nějaký spolehlivý typový systém přiřadí části programu jiný typ než *type_error*, potom při běhu cílového kódu vygenerovaného z této části programu nemůže nastat typová chyba. Jazyk je *přísně typovaný* (strongly typed), pokud jeho překladač může zaručit, že program, který příjme, se bude provádět bez typových chyb.

V praxi se však mohou některé kontroly provádět výlučně dynamicky. Například pokud nejprve deklarujeme

```
table: array [0..255] of char;
i: integer;
```

a potom počítáme `table[i]`, nemůže překladač obecně zaručit, že při provádění programu bude hodnota `i` ležet v intervalu 0 až 255. Pouze v některých programech lze pomocí technik analýzy toku dat zda je `i` v určitých mezích. Žádná technika to však nemůže provést správně ve všech případech.

### 7.1.3 Zotavení po chybě při typové kontrole

Vzhledem k tomu, že typová kontrola má schopnost zachycovat chyby v programech, je pro podsystém typové kontroly důležité, aby při výskytu chyby provedl něco rozumného. Nejdříve ze všeho musí překladač ohlásit podstatu a pozici chyby. Při typové kontrole vyžadujeme, aby došlo k zotavení a mohl se kontrolovat i zbytek programu. Zotavení musí být zabudováno již od počátku do typového systému.

Zavedení zpracování chyb může vést k typovému systému, který jde mnohem dále než systém nutný pouze ke specifikaci správných programů. Například nastala-li již chyba, nemůžeme znát typ nesprávně vytvořeného úseku programu. Zacházení s neúplnými informacemi vyžaduje techniky podobné metodám potřebným v jazycích, které nevyžadují deklaraci identifikátorů před jejich použitím. K zajištění konzistentního použití nedeklarovaných nebo zjevně nesprávně deklarovaných identifikátorů lze použít typových proměnných, představujících neznámý datový typ.

## 7.2 Ekvivalence typových výrazů

Během typové kontroly často vyžadujeme, aby dva datové typy byly ekvivalentní. Pojem ekvivalence datových typů však prozatím nebyl přesně definován; není například zřejmé, zda dva různě pojmenované typy se shodnou vnitřní strukturou jsou či nejsou ekvivalentní. V programovacích jazycích se setkáváme v podstatě se dvěma základními přístupy. *Ekvivalence podle jmen* považuje každý pojmenovaný typ za jedinečný, odlišný od všech ostatních pojmenovaných či nepojmenovaných typů; dva typové výrazy jsou ekvivalentní podle jména právě tehdy, jsou-li identické. Při zjišťování *ekvivalence podle struktury* nejprve nahradíme všechna jména odpovídajícími typovými výrazy; dva typové výrazy považujeme za ekvivalentní, jestliže po tomto nahrazení mají oba výrazy stejnou vnitřní strukturu.

**Příklad 7.2.** Uvažujme následující úsek deklarací v jazyce Pascal:

```
type link = ↑ cell;
var  next : link;
     last : link;
     p    : ↑ cell;
     q, r : ↑ cell;
```

Identifikátor `link` je zde jménem typu `↑cell`. Zajímá nás, zda typy proměnných `next`, `last`, `p`, `q` a `r` jsou či nejsou identické. Proměnným `next` a `last` je přiřazen typový výraz `link`, ostatním proměnným výraz *pointer*(`cell`). Je-li implementována ekvivalence podle jmen, mají proměnné `next` a `last` stejný typ, neboť jim odpovídající typové výrazy jsou identické. Podobně proměnné `p`, `q` a `r` mají stejný typ, ovšem odlišný od typu proměnné `next`. Uvažujeme-li však strukturální ekvivalenci, jsou typy všech proměnných stejné, neboť po nahrazení jména typu `link` odpovídajícím typovým výrazem *pointer*(`cell`) z jeho definice dostaneme pro všechny proměnné výrazy se stejnou vnitřní strukturou.

V některých implementacích se k ekvivalenci podle jmen přistupuje poněkud odlišným způsobem. Každému výskytu nepojmenovaného typu se přiřadí implicitní jméno, které tento výskyt odlišuje od všech ostatních výskytů téhož nepojmenovaného typu. V našem příkladě by tedy proměnná `p` mohla mít jiný typ než proměnné `q` a `r`. Tento přístup podstatně zjednodušuje implementaci ekvivalence typů, neboť pokud například reprezentujeme typy proměnných pomocí ukazatelů na datové struktury popisující konkrétní výskyt typu, můžeme za ekvivalentní datové typy považovat ty, které jsou reprezentovány stejnými ukazateli.

Pro testování strukturální ekvivalence můžeme použít algoritmu obdobnému tomu, který je uveden na obr. 7.3. Funkce $sequiv(s,t)$ vrátí hodnotu true, pokud jsou typové výrazy $s$ a $t$ strukturálně ekvivalentní, a hodnotu false v opačném případě.

> **function** sequiv(s,t): **boolean**;
> **begin**
>   **if** $s$ a $t$ jsou stejné základní typy **then**
>     **return true**
>   **else if** $s = array(s_1, s_2)$ **and** $t = array(t_1, t_2)$ **then**
>     **return** $sequiv(s_1, t_1)$ **and** $sequiv(s_2, t_2)$
>   **else if** $s = s_1 \times s_2$ **and** $t = t_1 \times t_2$ **then**
>     **return** $sequiv(s_1, t_1)$ **and** $sequiv(s_2, t_2)$
>   **else if** $s = pointer(s_1)$ **and** $t = pointer(t_1)$ **then**
>     **return** $sequiv(s_1, t_1)$
>   **else if** $s = s_1 \rightarrow s_2$ **and** $t = t_1 \rightarrow t_2$ **then**
>     **return** $sequiv(s_1, t_1)$ **and** $sequiv(s_2, t_2)$
>   **else**
>     **return false**
> **end**

Obrázek 7.3: Testování strukturální ekvivalence typových výrazů

V některých implementacích překladačů se pro kódování typových výrazů používají i jiné datové struktury než graf. Datový typ může být zakódován jako posloupnost bitů tvořená



kódem základního datového typu, ke kterému se přidávají kódy typových konstruktorů v pořadí jejich aplikace. Výhodou tohoto přístupu je úsporná reprezentace a jednodušší testování strukturální ekvivalence, neboť dva strukturálně odlišné datové typy nemohou mít stejnou bitovou reprezentaci. Naopak nevýhodou je omezení přípustné složitosti datových typů, které může programátor používat, obvykle délkou slova procesoru.

Při implementaci ekvivalence podle struktury musíme uvažovat i možnost rekurzivní definice typu — např. datový typ záznam může v sobě obsahovat ukazatel na jiný záznam téhož typu. Je-li datový typ v překladači reprezentován grafem, obdržíme po nahrazení jmen typů odpovídajícími grafy cyklický graf, a musíme tedy zajistit, aby se algoritmus zjišťující strukturální ekvivalenci typů choval korektně i v tomto případě.

## 7.3 Typové konverze

Uvažujme výraz `x+i`, kde `x` je typu real a `i` typu integer. Vzhledem k tomu, že reprezentace obou typů v počítači je odlišná a že počítač pro operace nad celými a reálnými čísly používá jiné instrukce, musí překladač nejprve zajistit konverzi jednoho z operandů na společný datový typ. To, zda tato konverze je implicitní nebo musí být explicitně zapsána programátorem, závisí na definici jazyka. Podobně musí být definována pravidla pro přiřazování hodnot do proměnných různých typů. Například v jazyce Pascal se při přiřazení celočíselné hodnoty do reálné proměnné provede implicitní konverze přiřazované hodnoty na typ real, ovšem při přiřazení reálného výrazu do celočíselné proměnné musí programátor explicitně definovat požadovanou konverzi voláním funkce *trunc* nebo *round*.

*Implicitní konverze* jednoho datového typu na druhý (často také zvané *koerce*) provádí překladač automaticky. Obvykle jsou tyto konverze omezeny na případy, kdy nemůže dojít ke ztrátě informace, např. konverze celého čísla na reálné. *Explicitní konverze* datových typů požaduje programátor obvykle ve formě volání určitých standardních funkcí nebo pomocí operátorů konverze. Například v jazyce Pascal funkce `ord` převádí znaky na celá čísla a funkce `chr` naopak celá čísla na znaky, zatímco v jazyce C se tato konverze provádí implicitně. V jazyce Ada jsou všechny konverze explicitní, čímž se zajistí skutečně důsledná typová kontrola a odhalení případných chyb v důsledku nesprávně zapsaných výrazů.

## 7.4 Přetěžování funkcí a operátorů

*Přetížený symbol* je takový, který má různý význam v závislosti na kontextu, ve kterém je použit. Ve výrazech je například přetížen symbol `+`, protože ve výrazu `A + B` může mít různý význam v závislosti na typech operandů `A` a `B`. V jazyce Ada jsou přetížené závorky (); výraz `A(I)` může být odkaz na `I`-tý prvek pole `A`, volání funkce `A` s parametrem `I` nebo explicitní konverze výrazu `I` na typ `A`.

Přetížení se nazývá *vyřešené*, pokud se nám podaří nalézt jednoznačný význam pro určitý výskyt přetíženého symbolu. U běžných programovacích jazyků, kde přetížení nastává pouze u standardních operátorů, není obvykle nalezení jednoznačného významu obtížné. V jazycích jako je Ada nebo C++ však může docházet k velmi komplikovaným situacím, kdy podvýraz nějakého výrazu může mít množinu možných typů a kdy pro vyřešení přetížení potřebujeme znát širší kontext.

**Příklad 7.3.** V jazyce Ada je jednou ze standardních interpretací operátoru `*` násobení dvou celých čísel. Tento operátor můžeme přetížit deklaracemi jeho dalších významů, např.

```
function "*" ( i, j : integer ) return complex;
function "*" ( x, y : complex ) return complex;
```

Po uvedených deklaracích množina možných typů operátoru `*` zahrnuje

$$\begin{array}{rcl} integer \times integer & \rightarrow & integer \\ integer \times integer & \rightarrow & complex \\ complex \times complex & \rightarrow & complex \end{array}$$

Za předpokladu, že konstanty `2`, `3` a `5` jsou pouze typu integer, může mít podvýraz `3*5` typ *integer* nebo *complex*, v závislosti na kontextu. Je-li úplný výraz `2*(3*5)`, musí být `3*5` typu *integer*, neboť operátor `*` může mít buď oba operandy typu *integer* nebo oba operandy typu *complex*. V jazyce C++ může být tato situace ještě komplikovaná tím, že programátor může definovat funkce pro implicitní konverzi typu *integer* na *complex*; tehdy by se po implicitní konverzi hodnoty `2` na typ *complex* mohl celý výraz vyhodnotit jako výraz typu *complex* a výsledný typ by byl opět nejednoznačný. Zpracování přetížených symbolů je obecně značně složitý problém; některé algoritmy, které se pro řešení přetížení používají, je možno nalézt v [2].

## 7.5 Polymorfické procedury a funkce

Obyčejné procedury a funkce umožňují provedení svého těla pouze s parametry pevných typů, které jsou uvedeny v deklaraci podprogramu nebo jsou dány implicitními konvencemi. Typy parametrů polymorfických procedur a funkcí naopak mohou být při každém volání podprogramu odlišné. V běžných programovacích jazycích se s polymorfismem setkáváme například u standardních operátorů pro indexování polí, volání funkcí a manipulaci s ukazateli. Například v jazyce C je-li ve výrazu `&x` operand `x` typu "...," je výsledek typu "ukazatel na ...." Za symbol "..." můžeme dosadit libovolný typ, takže operátor `&` je v jazyce C polymorfický.

Polymorfické procedury a funkce jsou z hlediska programátorského velmi efektivním prostředkem pro vyjadřování obecných algoritmů. Například potřebujeme-li v Pascalu funkci pro zjištění délky seznamu celých nebo reálných čísel, musíme stejný algoritmus zapsat dvakrát, přičemž lišit se budou pouze deklarace typu parametru funkce. Výhodnější by bylo použít polymorfické funkce, která by umožňovala výpočet délky seznamu prvků libovolného typu (který ve vlastním výpočtu nehraje žádnou roli).

Abychom mohli specifikovat typy polymorfických funkcí, musíme v typových výrazech použít *typové proměnné*. Typové proměnné budeme označovat písmeny řecké abecedy $\alpha, \beta, \ldots$ a budou reprezentovat vždy konkrétní neznámý typ. Například operátor `&` jazyka C bude mít typ

$$\alpha \rightarrow pointer(\alpha)$$

.

V programovacích jazycích, které nevyžadují explicitní definice typů proměnných a funkcí (například ve funkcionálních jazycích jako je jazyk ML), musíme typy jednotlivých jazykových konstrukcí určovat na základě kontextu. Tento proces se nazývá *inference typů*. Například ve funkci



```
fun length(lptr) =
        if null(lptr) then 0
        else length(tl(lptr)) + 1;
```

se na druhém řádku volá standardní funkce `null`, která je typu $list(\alpha) \rightarrow boolean$, kde *list* je konstruktor seznamu. Odtud je zřejmé, že `lptr` musí být typu $list(\alpha)$, kde $\alpha$ je nějaký (libovolný) typ. Jako výsledek se vrací celočíselná konstanta `0`, proto je výsledek funkce `length` typu *integer*. Funkce `length` má tedy typ $list(\alpha) \rightarrow integer$. Na třetím řádku můžeme už jenom provést na základě znalosti typu funkcí `tl` a `length` kontrolu, zda je uvedený výraz typově správný.

Inference typů lze využít i v překladačích klasických jazyků pro doplňování chybějících informací v době překladu. Například v jazyce C můžeme z volání funkce odvodit typy jejích operandů a výsledku a později, v okamžiku její definice, zkontrolovat, zda je tato definice konzistentní s předchozími voláními.

### 7.5.1 Unifikace typových výrazů

Při inferenci typů je základním problémem nalezení společné instance dvou typových výrazů s typovými proměnnými — jejich *unifikace*. Unifikaci můžeme definovat pomocí funkce $S$ zvané *substituce*, která proměnným přiřazuje výrazy. Zápis $S(e)$ představuje výraz získaný tak, že všechny proměnné $\alpha$ obsažené v $e$ nahradíme hodnotou $S(\alpha)$. Potom $S$ je unifikátorem pro $e$ a $f$, právě když $S(e) = S(f)$.

Máme-li dva typové výrazy $e$ a $f$, hledáme takovou nejobecnější substituci proměnných v nich obsažených, aby po této substituci oba výrazy byly ekvivalentní. Výsledkem unifikace může být buď tato substituce, nebo zjištění, že společná instance výrazů neexistuje. Speciálním případem unifikace je testování ekvivalence dvou typových výrazů; pokud výrazy $e$ a $f$ neobsahují proměnné, je možné je unifikovat právě tehdy, jestliže jsou ekvivalentní.

**Příklad 7.4.** Uvažujme následující dva typové výrazy:

$$\begin{array}{rcl} ((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_3)) & \rightarrow & list(\alpha_2) \\ ((\alpha_3 \rightarrow \alpha_4) \times list(\alpha_3)) & \rightarrow & \alpha_5 \end{array}$$

Pro tyto výrazy můžeme najít substituci $S$ takovou, že $S(\alpha_1) = S(\alpha_3) = \alpha_3$, $S(\alpha_2) = S(\alpha_4) = \alpha_2$, $S(\alpha_5) = list(\alpha_2)$, která zobrazuje $e$ a $f$ na výraz

$$S(e) = S(f) = ((\alpha_3 \rightarrow \alpha_2) \times list(\alpha_3)) \rightarrow list(\alpha_2)$$

■

Jeden z možných unifikačních algoritmů je uveden v [2]; podobné algoritmy se používají při vyhodnocování programů v logických programovacích jazycích (např. Prolog) nebo obecně při řešení problémů z oblasti umělé inteligence.

# Kapitola 8

# Generování intermediárního kódu

V analyticko-syntetickém modelu překladu převádí přední část překladače zdrojový program do intermediární reprezentace, ze které dále zadní část překladače generuje cílový kód. Je samozřejmě možné — a také se tak často postupuje — přeložit zdrojový program přímo do cílového jazyka. Překlad využívající nějakého strojově nezávislého mezikódu má však své výhody:

1. Zjednodušuje se přepracování překladače pro jiný cílový jazyk (retargeting). Stačí vytvořit pouze novou koncovou část.
2. Mezikód lze optimalizovat s využitím metod strojově nezávislé optimalizace.

V této kapitole si ukážeme použití metod syntaxí řízeného překladu pro překlad základních programových konstrukcí jako jsou deklarace, přiřazení a řídicí příkazy do intermediárního kódu. Většina uvedených metod se dá použít během překladu zdola nahoru nebo shora dolů, takže generování intermediárního kódu se dá podle potřeby začlenit do syntaktické analýzy.

## 8.1 Intermediární jazyky

Jako intermediární reprezentace programu se používají nejčastěji stromy (případně obecné grafy) a zásobníkový nebo tříadresový kód. Výběr mezikódu je často dán požadavky na efektivitu jeho dalšího zpracování. Například pro rozsáhlejší optimalizace je výhodnější použít tříadresového kódu místo zásobníkového. Naopak zásobníkový kód může být výhodnější v překladačích generujících kód pro počítače se zásobníkovou architekturou.

### 8.1.1 Grafová reprezentace

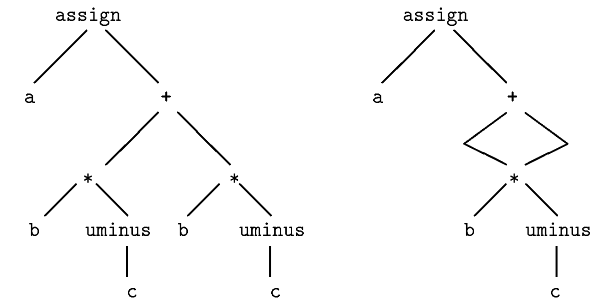
Za přirozenou grafovou reprezentaci programu můžeme považovat přímo syntaktický strom nebo DAG. Na obr. 8.2 je znázorněn strom a DAG pro přiřazovací příkaz `a := b * -c + b * -c`.

Pomocí grafu se často v překladači reprezentují deklarace, které se neobjevují přímo v mezikódu (viz odstavec 5.1), a výrazy, jejichž kód se někdy může v mezikódu vyskytovat na jiném místě, než kde byl výraz uveden ve zdrojovém programu. Například pro příkaz cyklu `for` jazyka C

```
for(p=first; p; p=p->next) print(p);
```



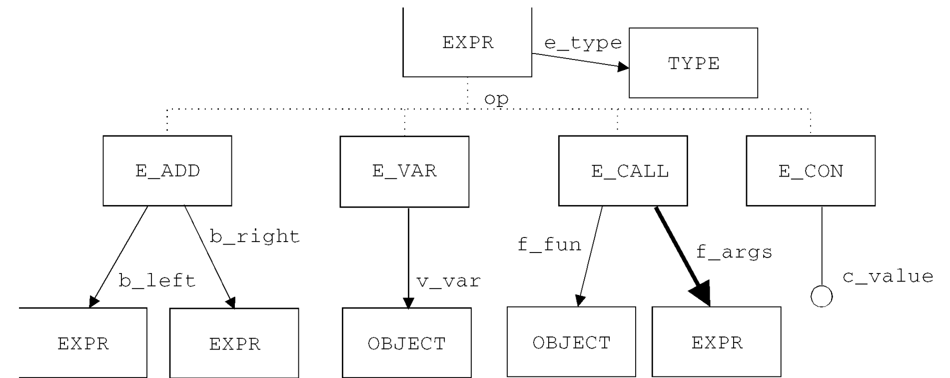





(a) Syntaktický strom (b) DAG

Obrázek 8.1: Grafická reprezentace výrazu `a := b * -c + b * -c`

se kód pro vyhodnocení výrazu `p=p->next` může vygenerovat až za konec těla cyklu a je tedy nutné nějakým způsobem uchovat výraz až do okamžiku, kdy bude tělo cyklu zpracováno. Překlad výrazu může probíhat dvoufázově: nejprve se vytvoří jeho grafová reprezentace, a pak se tento graf ve vhodném okamžiku převede například do tříadresového kódu.

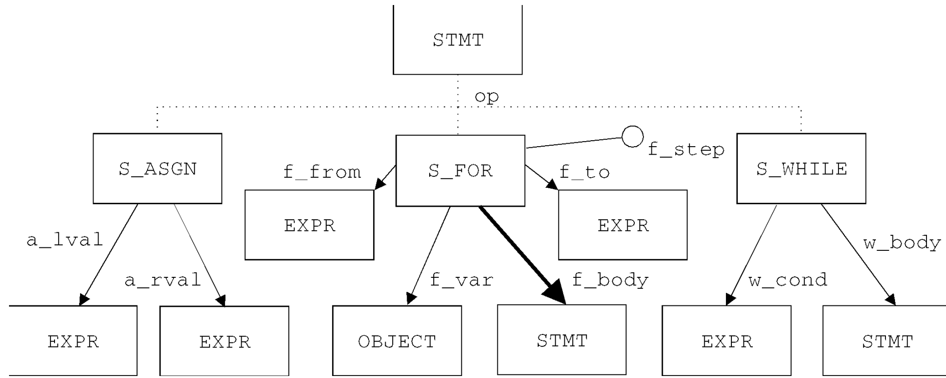



Obrázek 8.2: E–R model výrazu

Pro obecnou reprezentaci jak výrazů, tak i příkazů programu můžeme rovněž použít E–R modelu z článku 5.1. Část sémantického grafu pro typické výrazy je uvedena na obr. 8.2, na obr. 8.3 je znázorněna struktura některých příkazů jazyka Pascal.

### 8.1.2 Zásobníkový kód

*Postfixová notace*, ze které vychází zásobníkový kód, představuje linearizovaný zápis syntak-




Obrázek 8.3: E–R model příkazu

```
VAR  b        ; ... (b)
VAR  c        ; ... (b) (c)
INV           ; ... (b) (-c)
MUL           ; ... (b * -c)
VAR  b        ; ... (b * -c) (b)
VAR  c        ; ... (b * -c) (b) (c)
INV           ; ... (b * -c) (b) (-c)
MUL           ; ... (b * -c) (b * -c)
ADD           ; ... (b * -c + b * -c)
ASG  a        ; ...
```

Obrázek 8.4: Zásobníkový kód pro výraz `a := b * -c + b * -c`

tického stromu; je to seznam uzlů, ve kterém je uzel stromu uveden vždy bezprostředně za svými přímými následníky. Postfixový zápis syntaktického stromu z obr. 8.2(a) je

```
a b c uminus * b c uminus * + assign
```

Postfixová notace neobsahuje explicitně hrany syntaktického stromu. Ty se dají zpětně odvodit z pořadí uzlů a z počtu jejich operandů.

*Zásobníkový kód* je tvořen posloupností příkazů, které obecně definují posloupnost akcí nad zásobníkem. Každá z těchto akcí představuje buď vložení hodnoty proměnné nebo konstanty na vrchol zásobníku, provedení určité operace nebo uložení hodnoty ze zásobníku do proměnné. Operandy a výsledky operací jsou obvykle uloženy na zásobníku. Příkaz `a := b * -c + b * -c` můžeme v zásobníkovém kódu zapsat například tak, jak ukazuje obr. 8.4. V poznámce je u každé instrukce zásobníkového kódu uveden obsah zásobníku po jejím provedení. Pořadí operandů a operátorů je stejné jako v postfixové notaci, ovšem postfixová notace operandy od operátorů formálně nerozlišuje.



### 8.1.3 Tříadresový kód

Tříadresový kód je posloupnost příkazů, které mají obecně tvar

`x := y` *op* `z`

kde `x`, `y` a `z` jsou jména, konstanty nebo překladačem vytvořené dočasné objekty; *op* představuje libovolný operátor, např. některý z aritmetických nebo logických operátorů. V operandech nemohou být žádné další výrazy, příkaz obsahuje vždy jen jediný operátor. Proto musí být složitější výrazy rozloženy na své nejjednodušší složky s použitím dočasných proměnných vytvořených překladačem. Zde je vidět zásadní rozdíl mezi zásobníkovým a tříadresovým kódem. Zásobníkový kód se odkazuje na operandy implicitně, na základě jejich pozice, zatímco tříadresový kód všechny operandy pojmenovává. Tím se značně zjednodušují optimalizace tříadresového mezikódu, při nichž se mohou jednotlivé příkazy navzájem libovolně přesouvat.

Pojmenování kódu vychází z toho, že každý příkaz obvykle obsahuje tři adresy, dvě pro operandy a jednu pro výsledek. Při implementaci mohou tyto adresy znamenat například ukazatele do tabulky symbolů na příslušné objekty.

Tříadresový kód je linearizovanou reprezentací syntaktického stromu nebo DAG, ve které jména generovaná překladačem odpovídají vnitřním uzlům grafu. Syntaktický strom a DAG z obr. 8.1 jsou na obr. 8.5 zapsány v tříadresovém kódu. Jména proměnných se mohou v zápisu používat přímo, proto zde nejsou žádné příkazy, které by reprezentovaly listy původního grafu.

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a  := t5
```

(a) Kód pro syntaktický strom

```
t1 := - c
t2 := b * t1
t5 := t2 + t2
a  := t5
```

(b) Kód pro DAG

Obrázek 8.5: Tříadresový kód pro strom a DAG z obr. 8.1

#### Typy příkazů tříadresového kódu

Příkazy tříadresového kódu jsou podobné příkazům jazyka asembleru. Mohou být označeny symbolickým návěštím, které se využívá v příkazech pro změnu toku řízení. Transformace symbolického jména na index příkazu v jeho vnitřní reprezentaci se provádí buď v samostatném průchodu, nebo metodou backpatching, kterou se budeme zabývat v odstavci 8.7.

V dalším textu budeme používat následující nejčastější tříadresové příkazy:

- Přiřazovací příkazy ve tvaru `x := y` *op* `z`, kde *op* je binární aritmetický nebo logický operátor.
- Přiřazovací příkazy ve tvaru `x :=` *op* `y`, kde *op* je unární operátor (unární minus, logická negace, operátory pro konverzi datových typů apod.).
- Kopírovací příkazy ve tvaru `x := y`.

- Nepodmíněný skok `goto L`.
- Podmíněné skoky ve tvaru `if x` *relop* `y goto L`, které se provedou tehdy, je-li splněna relace *op* mezi hodnotami `x` a `y`.
- Příkazy `param x` a `call p,`$n$ pro volání procedury a `return y` s volitelnou hodnotou `y` reprezentující návratovou hodnotu. Typická posloupnost těchto příkazů pro volání procedury `p(`$x_1$`, `$x_2$`, ..., `$x_n$`)` je

  ```
  param x1
  param x2
     ...
  param xn
  call p,n
  ```

  kde $n$ je počet skutečných parametrů předávaných proceduře.
- Přiřazení s indexováním ve tvaru `x:=y[i]` nebo `x[i]:=y`.
- Přiřazení adres a nepřímý přístup přes ukazatel ve tvaru `x:=&y`, `x:=*y` a `*x:=y`. První z těchto příkazů uloží do `x` adresu objektu `y`, další uloží do `x` hodnotu, jejíž adresa je v proměnné `y` a poslední uloží na adresu, která je v proměnné `x` hodnotu `y`.

Výběr operátorů je velmi důležitou součástí návrhu intermediárního kódu. Soubor operátorů musí být dostatečně bohatý, aby se jím daly vyjádřit všechny operace zdrojového jazyka. Menší počet operátorů zjednodušuje implementaci generátoru kódu, avšak vede k podstatně delším úsekům mezikódu, které se dále musí optimalizovat.

**Implementace tříadresových příkazů**

Tříadresové příkazy jsou abstraktní formou intermediárního kódu. V překladači se tyto příkazy mohou implementovat jako záznamy s položkami pro operátor a operandy. Obvykle se pro ně používá jedna z následujících reprezentací:

- *Čtveřice (quadruples).* Čtveřice je struktura se čtyřmi položkami, které označíme *op*, $arg1$, $arg2$ a $result$. Položka *op* obsahuje kód operátoru, $arg1$ a $arg2$ operandy a $result$ výsledek. Některé příkazy nemusejí využívat všechny položky, např. unární operátory nevyužívají $arg2$.
- *Trojice (triples).* V této reprezentaci datová struktura reprezentující příkaz neobsahuje položku pro výsledek. Výsledek je v operandech dalších příkazů reprezentován číslem příslušné trojice.
- *Nepřímé trojice (indirect triples).* Nevýhodou předchozí reprezentace je, že se jednotlivé trojice nemohou jednoduše přesouvat nebo rušit, například během optimalizace kódu. Proto se může využít ještě dalšího pomocného pole, které obsahuje pouze ukazatele na jednotlivé trojice a které definuje jejich skutečné pořadí.



| | *op* | *arg1* | *arg2* | *result* |
|---|---|---|---|---|
| (0) | `uminus` | `c` | | `t1` |
| (1) | `*` | `b` | `t1` | `t2` |
| (2) | `uminus` | `c` | | `t3` |
| (3) | `*` | `b` | `t3` | `t4` |
| (4) | `+` | `t2` | `t4` | `t5` |
| (5) | `assign` | `t5` | | `a` |

(a) Čtveřice

| | *op* | *arg1* | *arg2* |
|---|---|---|---|
| (0) | `uminus` | `c` | |
| (1) | `*` | `b` | (0) |
| (2) | `uminus` | `c` | |
| (3) | `*` | `b` | (2) |
| (4) | `+` | (0) | (3) |
| (5) | `assign` | `a` | (4) |

(b) Trojice

Obrázek 8.6: Reprezentace tříadresových příkazů trojicemi a čtveřicemi

| | *příkaz* |
|---|---|
| (0) | (14) |
| (1) | (15) |
| (2) | (16) |
| (3) | (17) |
| (4) | (18) |
| (5) | (19) |

| | *op* | *arg1* | *arg2* |
|---|---|---|---|
| (14) | `uminus` | `c` | |
| (15) | `*` | `b` | (14) |
| (16) | `uminus` | `c` | |
| (17) | `*` | `b` | (16) |
| (18) | `+` | (15) | (17) |
| (19) | `assign` | `a` | (18) |

Obrázek 8.7: Reprezentace tříadresových příkazů nepřímými trojicemi

## 8.2 Deklarace

### 8.2.1 Deklarace proměnných

Při zpracování deklarací je základním úkolem překladače vytváření tabulky symbolů. S tím obvykle souvisí i shromažďování informací o datových typech a velikostech jejich reprezentace a přidělování adres proměnným a složkám záznamů, což již není nezávislé na generovaném cílovém kódu. Překladač musí uvažovat nejen konkrétní velikosti objektů různých typů, ale také další požadavky definované architekturou cílového počítače, například zarovnávání. Při vytváření přemístitelného kódu je adresa objektu definována vždy dvěma údaji: příslušností do určité samostatně adresované skupiny objektů (např. globální a lokální proměnné nebo procedury, konstanty, externí proměnné) a relativní adresou vzhledem k začátku této skupiny. Pro každou takovou skupinu můžeme udržovat samostatný čítač adres, který se při deklaraci objektu patřícího do příslušné skupiny vždy zvýší o velikost datového typu objektu.

**Příklad 8.1.** Překladové schéma na obr. 8.8 popisuje překlad posloupnosti deklarací ve tvaru **id**: $T$. Současná relativní adresa pro deklarované proměnné je uložena v proměnné *offset* a je na začátku nastavena na nulu. Procedura $enter(name, type, offset)$ vytvoří novou položku tabulky symbolů pro proměnnou $name$ typu $type$, které bude přidělena relativní adresa *offset*. Syntetizované atributy $type$ a $width$ nonterminálu $T$ představují typ a jeho velikost. Typ je reprezentován grafem, jehož uzly se vytvářejí ze základních typů $integer$ a $real$ funkcemi $array$ a $pointer$. Předpokládáme, že hodnoty typu $integer$ a ukazatele vyžadují 4 slabiky a hodnoty typu $real$ 8 slabik paměti. ■

Inicializace proměnné *offset* ve schématu na obr. 8.8 má tvar

$$P \rightarrow \{offset := 0\}D \qquad (8.1)$$

| | |
|---|---|
| $P \rightarrow$ D | $\{ \; offset := 0 \; \}$ |
| $D \rightarrow D \; ; \; D$ | |
| $D \rightarrow \textbf{id}: T$ | $\{ \; enter(\textbf{id}.name, T.type, offset);$ $offset := offset + T.width \; \}$ |
| $T \rightarrow \textbf{integer}$ | $\{ \; T.type := integer;$ $T.width := 4 \; \}$ |
| $T \rightarrow \textbf{real}$ | $\{ \; T.type := real;$ $T.width := 8 \; \}$ |
| $T \rightarrow \textbf{array} \; [ \; \textbf{num} \; ] \; \textbf{of} T_1$ | $\{ \; T.type := array(\textbf{num}.val, T_1.type);$ $t.width := \textbf{num}.val \times T_1.width \; \}$ |
| $T \rightarrow \uparrow T_1$ | $\{ \; T.type := pointer(T_1.type);$ $T.width := 4 \; \}$ |

Obrázek 8.8: Výpočet typů a relativních adres v deklaracích

Pomocí nonterminálů generujících prázdný řetězec (markerů) můžeme taková pravidla přepsat do tvaru, kdy jsou všechny akce na konci pravidel. Např. s využitím nonterminálu $M$ přepíšeme (8.1) na

$$P \rightarrow M \; D$$
$$M \rightarrow \epsilon \qquad \{offset := 0\}$$

Přesun akcí na konec pravidel umožňuje provádět překlad zdola nahoru, kdy se sémantické akce provádějí během redukce pravé strany pravidla.

### 8.2.2 Deklarace v jazycích s blokovou strukturou

V jazycích jako je Pascal nebo C mohou být jednotlivé bloky deklarací do sebe zanořené. Na začátku zanořeného bloku deklarací se dočasně potlačí zpracování deklarací nadřazeného bloku, ve kterém se pokračuje až po uzavření zanořeného bloku. V kapitole 5 jsme pro tento účel zavedli operace `tabopen` a `tabclose`, které otevíraly a zavíraly jednu úroveň blokově strukturované tabulky symbolů. Následující příklad ukazuje, jak se bloková struktura jazyka odrazí ve zpracování deklarací.

**Příklad 8.2.** Jazyk deklarací z příkladu 8.1 rozšíříme o pravidlo

$$D \rightarrow \textbf{proc id} \; ; \; D \; ; \; S$$

umožňující deklarovat proceduru s lokálními deklaracemi. Na začátku vnořeného bloku deklarací musíme nejprve uschovat současnou hodnotu čítače *offset* (použijeme k tomu atributu markeru $M$), nastavit tento čítač na nulu a otevřít novou úroveň tabulky symbolů. Po ukončení těla bloku naopak uzavřeme současnou úroveň a obnovíme původní hodnotu čítače viz obr. 8.9. ■

Jazyk C sice neumožňuje do sebe vkládat deklarace funkcí, avšak dovoluje do sebe zanořovat bloky deklarací proměnných. Všechny proměnné v zanořených blocích spolu sdílejí společnou oblast paměti; jejich relativní adresy se počítají od začátku oblasti lokálních proměnných funkce, v níž jsou deklarované. To znamená, že při vstupu do bloku musíme nechat



| | |
|---|---|
| $D \rightarrow \textbf{proc id}; \; M \; D \; ; \; S$ | $\{ \; tabclose();$ $offset := M.offset \; \}$ |
| $M \rightarrow$ | $\{ \; M.offset = offset;$ $offset = 0;$ $tabopen() \; \}$ |

Obrázek 8.9: Zpracování zanořených deklarací

hodnotu *offset* beze změny. Při výstupu z bloku můžeme obnovit původní hodnotu a případně tak využít uvolněné paměti pro další proměnné.

Podobným způsobem jako lokální proměnné se zpracovávají také deklarace položek záznamů. Na začátku deklarace záznamu se rovněž otevře nová úroveň tabulky symbolů a vynuluje se čítač adres, při ukončení záznamu se však musí deklarace položek, které se při zpracování těla záznamu uložily do tabulky, uchovat jako atribut datového typu záznam. Tyto deklarace se totiž budou dále používat při odkazech na složky záznamu ve výrazech. Nejjednodušší implementace úschovy položek záznamu je při použití tabulky symbolů strukturované jako zásobník stromů — uschová se ukazatel na kořen stromu pro poslední otevřenou úroveň tabulky. Deklarace položek záznamů s variantami (v Pascalu) nebo unií (v jazyce C) probíhá obdobně, pouze se po deklaraci nové složky nezvyšuje čítač adres a tím se všem odpovídajícím položkám přidělí totéž místo. Pouze je třeba sledovat délku největší položky, která se stane délkou celého datového typu.

S deklaracemi položek záznamů také souvisí zpracování příkazu **with** jazyka Pascal. Tento příkaz zpřístupní současně všechny složky určitého záznamu. Příkaz **with** se dá implementovat tak, že znovu otevřeme novou úroveň deklarací a vložíme do ní část tabulky symbolů, kterou jsme uschovali při dokončení deklarace záznamu. Po ukončení platnosti příkazu **with** opět tuto úroveň zrušíme.

## 8.3 Přiřazovací příkazy a výrazy

Pro překlad celočíselných aritmetických výrazů a přiřazení do jednoduchých proměnných můžeme použít schématu z obr. 8.10. V tomto schématu se používá funkce *lookup* pro vyhledání proměnné v tabulce symbolů na základě jejího jména; pokud se jméno v tabulce nenajde, funkce vrátí hodnotu *nil*. Funkce *newtemp* vrátí ukazatel na nově vytvořenou dočasnou proměnnou. Dočasné proměnné mohou být obecně uloženy rovněž v tabulce symbolů, pokud jim přidělíme speciální jména, která nemohou být použita programátorem pro proměnné v programu.

Sémantické akce na obr. 8.10 používají pro výstup tříadresových příkazů procedury *emit*, jejíž parametry uvádíme poněkud zjednodušeně buď jako řetězcové konstanty, nebo jako jména atributů, jejichž příslušné hodnoty se mají předat na výstup.

Uvedené překladové schéma se dá použít i v případě, že pracujeme s jazykem, který má blokovou strukturu, neboť jediná změna nastane v implementaci funkce *lookup* (viz kapitola 5). Dočasné proměnné se považují vždy za lokální proměnné podprogramu, ve kterém jsou použity.

$$
\begin{array}{lll}
S \to \textbf{id} := E & \{ & p := lookup(\textbf{id}.name); \\
 & & \textbf{if } p \neq nil \textbf{ then} \\
 & & \quad emit(p\ ':='\ E.place) \\
 & & \textbf{else } error\ \} \\
E \to E_1\ +\ E_2 & \{ & E.place := newtemp; \\
 & & emit(E.place\ ':='\ E_1.place\ '+'\ E_2.place)\ \} \\
E \to E_1\ *\ E_2 & \{ & E.place := newtemp; \\
 & & emit(E.place\ ':='\ E_1.place\ '*'\ E_2.place)\ \} \\
E \to -\ E_1 & \{ & E.place := newtemp; \\
 & & emit(E.place\ ':='\ \texttt{'uminus'}\ E_1.place)\ \} \\
E \to (\ E_1\ ) & \{ & E.place := E_1.place\ \} \\
E \to \textbf{id} & \{ & p := lookup(\textbf{id}.name); \\
 & & \textbf{if } p \neq nil \textbf{ then} \\
 & & \quad E.place := p \\
 & & \textbf{else } error\ \}
\end{array}
$$

Obrázek 8.10: Překladové schéma pro překlad aritmetických výrazů a přiřazení

### 8.3.1 Přidělování dočasných proměnných

Pro přidělování dočasných proměnných se dají použít dvě odlišné strategie. Pro optimalizující překladače je výhodné, pokud každé volání *newtemp* vrátí nové jméno, odlišné od všech předchozích. To však může mít za následek přeplňování tabulky symbolů (nebo obecně pracovní paměti) informacemi, které se používají jen velmi krátce.

Další možností, která se využívá zejména u jednoprůchodových překladačů, je vícenásobné využívání dočasných proměnných. Ze schematu na obr. 8.10 je zřejmé, že např. překladem výrazu $E_1\ +\ E_2$ vznikne kód ve tvaru

vypočti $E_1$ do proměnné `t1`
vypočti $E_2$ do proměnné `t2`
`t := t1 + t2`

po jehož vyhodnocení již nejsou proměnné `t1` a `t2` dále potřebné. Doba života všech dočasných proměnných použitých pro vyhodnocení $E_1$ je vlastně zanořena do doby života proměnné `t`, takže je možné upravit funkci *newtemp* tak, že pro přidělování dočasných proměnných využívá zásobníku. Rovněž je možné do schematu zařadit explicitní volání procedury pro uvolnění dočasné proměnné; tato proměnná se zařadí do seznamu volných dočasných proměnných, odkud se pak může znovu použít při dalším volání *newtemp*.

Přidělování dočasných proměnných je poněkud komplikovanější, pokud jim může být přiřazena hodnota více než jedenkrát, např. v podmíněném výrazu `a > b ? a : b` v jazyce C se musí hodnoty obou větví dostat do téže proměnné. Podobný problém nastává tehdy, pokud provádíme optimalizaci společných podvýrazů; tehdy se může hodnota jedné dočasné proměnné používat na více místech.

### 8.3.2 Adresování prvků polí

Pole obsahují vždy prvky stejného typu, které se mohou umístit bezprostředně jeden za druhým do společného bloku paměti. Je-li velikost každého prvku $w$, je $i$-tý prvek pole `A` uložen



na adrese

$$base + (i - low) \times w \tag{8.2}$$

kde *low* je dolní mez indexu pole a *base* je relativní adresa přidělené oblasti paměti (neboli relativní adresa prvku `A[`*low*`]`). Výraz (8.2) můžeme přepsat do tvaru, který umožňuje jeho částečné vyhodnocení již v době překladu:

$$i \times w + (base - low \times w)$$

Podvýraz $c = base - low \times w$ se dá vypočítat v okamžiku deklarace pole a uložit do tabulky symbolů pro `A`. Při generování kódu pro přístup k prvku `A[`$i$`]` získáme jeho relativní adresu jednoduše přičtením $i \times w$ k $c$.

Stejnou úvahu můžeme provést pro vícerozměrná pole. Dvojrozměrná pole se obvykle ukládají v paměti po řádcích, kdy můžeme pro výpočet relativní adresy prvku `A[`$i_1$`,`$i_2$`]` použít výrazu

$$base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$$

kde $low_1$ a $low_2$ jsou dolní meze indexů $i_1$ a $i_2$ a $n_2$ je počet sloupců pole, tj. $n_2 = high_2 - low_2 + 1$), kde $high_2$ je horní mez indexu $i_2$. Uvedený výraz můžeme opět přepsat do tvaru, ve kterém je oddělena konstantní a proměnná část adresy, jako

$$((i_1 \times n_2) + i_2) \times w + (base - ((low_1 \times n_2) + low_2) \times w) \tag{8.3}$$

Druhý operand tohoto součtu může být vypočten již v době překladu.

Zobecněním výrazu 8.3 pro $k$-rozměrné pole uložené tak, že se poslední index mění nejrychleji, dostaneme pro relativní adresu prvku `A[`$i_1$`,`$i_2$`,`$\ldots$`,`$i_k$`]` následující výraz (*mapovací funkci*):

$$\begin{aligned} &((\cdots((i_1 n_2 + i_2) n_3 + i_3) \cdots) n_k + i_k) \times w \\ &+ base - ((\cdots((low_1 n_2 + low_2) n_3 + low_3) \cdots) n_k + low_k) \times w \end{aligned} \tag{8.4}$$

Vzhledem k tomu, že pro $j$-tý index předpokládáme pevnou hodnotu $n_j = high_j - low_j + 1$), můžeme výraz na druhém řádku v (8.4) vypočítat v době překladu a uložit do položky tabulky symbolů pro `A`. V jazyce C je celý výpočet jednodušší, neboť dolní mez všech indexů je vždy nulová, takže konstantní část výrazu (8.4) je vždy rovna pouze *base*.

Překladové schéma na obr. 8.11 popisuje překlad přiřazovacích příkazů s aritmetickými výrazy a indexy. Toto schéma přímo implementuje výpočet podle vztahu (8.4). Oproti schématu z obr. 8.10 je operandem, resp. levou stranou přiřazení místo symbolu **id** nonterminál $L$ představující $l$-hodnotu (tj. hodnotu, která může stát na levé straně přiřazení). Pro tento nonterminál můžeme zavést následující pravidla:

$$\begin{array}{rcl} L & \to & \textbf{id}\ [\ Elist\ ]\ |\ \textbf{id} \\ Elist & \to & Elist\ ,\ E\ |\ E \end{array}$$

Pro vlastní výpočet je však výhodnější tato pravidla přepsat do tvaru, kdy máme během zpracování indexů již k dispozici ukazatel na položku tabulky symbolů pro indexované pole:

$$\begin{array}{rcl} L & \to & Elist\ ]\ |\ \textbf{id} \\ Elist & \to & Elist\ ,\ E\ |\ \textbf{id}\ [\ E \end{array}$$

Ve schematu na obr. 8.11 se tento ukazatel předává jako atribut *Elist.array*. Dále se pro počet dimenzí (indexových výrazů) používá atributu *Elist.ndim*. Funkce $limit(array, j)$ vrací hodnotu $n_j$, počet prvků v $j$-té dimenzi pole, na jehož záznam v tabulce symbolů ukazuje *array*. Funkce $c(array)$ vrátí konstantní část výrazu (8.4) pro pole *array*. Atribut *Elist.place* obsahuje jméno dočasné proměnné, do které byla uložena hodnota vypočtená z indexových výrazů v *Elist*.

```
(1) S → L := E           { if L.offset = null then /* L je jednoduchá proměnná */
                                emit(L.place ':=' E.place);
                           else
                                emit(L.place '[' L.offset ']' ':=' E.place) }
(2) E → E1 + E2          { E.place := newtemp;
                           emit(E.place ':=' E1.place '+' E2.place)}
(3) E → ( E1 )           { E.place := E1.place }
(4) E → L                { if L.offset = null then /* L je jednoduchá proměnná */
                                E.place := L.place
                           else begin
                                E.place := newtemp;
                                emit(E.place ':=' L.place '[' L.offset ']')
                           end }
(5) L → Elist ]          { L.place := newtemp;
                           L.offset := newtemp;
                           emit(L.place ':=' c(Elist.array));
                           emit(L.offset ':=' Elist.place '*' width(Elist.array)) }
(6) L → id               { L.place := id.place;
                           L.offset := null }
(7) Elist → Elist1 , E   { t := newtemp;
                           m := Elist1.ndim + 1;
                           emit(t ':=' Elist1.place '*' limit(Elist1.array, m));
                           emit(t ':=' t '+' E.place);
                           Elist.array := Elist1.array;
                           Elist.place := t;
                           Elist.ndim := m }
(8) Elist → id [ E       { Elist.array := id.place;
                           Elist.place := E.place;
                           Elist.ndim := 1 }
```

Obrázek 8.11: Překladové schéma pro výrazy s indexy

Nonterminál $L$, reprezentující $l$-hodnotu, má dva atributy, *L.place* a *L.offset*. Je-li $L$ jednoduchá proměnná, obsahuje *L.place* ukazatel na příslušnou položku tabulky symbolů a *L.offset* je **null**. V opačném případě ukazuje *L.place* na položku tabulky symbolů pro pole a *L.offset* na položku pro dočasnou proměnnou, do které byla uložena vypočtená relativní adresa prvku pole.

**Příklad 8.3.** Nechť `A` je pole $10 \times 20$ s dolními mezemi indexů $low_1 = low_2 = 1$. Počty prvků v jednotlivých dimenzích jsou tedy $n_1 = 10$ a $n_2 = 20$. Nechť velikost prvku $w$ je 4. Přiřazení `x := A[y,z]` se přeloží do následující posloupnosti tříadresových příkazů:



```
t1 := y * 20
t1 := t1 + z
t2 := c          /* konstanta c = base_A − 84 */
t3 := 4 * t1
t4 := t2[t3]
x  := t4
```

V příkladu jsme použili místo atributu **id**.*place* přímo jméno proměnné. ■

### 8.3.3 Konverze typů během přiřazení

V uvedených příkladech překladových schémat jsme zatím uvažovali pouze aritmetické výrazy tvořené operandy téhož typu. V praxi se však běžně pracuje se smíšenými výrazy, u nichž musí překladač buď vygenerovat příslušné implicitní typové konverze, nebo musí nahlásit chybu.

```
E.place := newtemp;
if E1.type = integer and E2.type = integer then begin
     emit(E.place ':=' E1.place 'int +' E2.place);
     E.type := integer
end
else if E1.type = real and E2.type = real then begin
     emit(E.place ':=' E1.place 'real +' E2.place);
     E.type := real
end
else if E1.type = integer and E2.type = real then begin
     u := newtemp;
     emit(u ':=' 'inttoreal' E1.place);
     emit(E.place ':=' u 'real +' E2.place);
     E.type := real
end
else if E1.type = real and E2.type = integer then begin
     u := newtemp;
     emit(u ':=' 'inttoreal' E2.place);
     emit(E.place ':=' E1.place 'real +' u);
     E.type := real
end
else
     E.type := type_error;
```

Obrázek 8.12: Sémantická akce pro pravidlo $E \rightarrow E_1 + E_2$

Uvažujme například jednoduché rozšíření aritmetického výrazu o datové typy *real* a *integer* s možností implicitní konverze celočíselného operandu na reálný ve smíšených výrazech. K tomu musíme zavést nový atribut *E.type*, jehož hodnota *real* nebo *integer* reprezentuje typ příslušného výrazu. Sémantická pravidla ve schématech na obr. 8.10 a 8.11 musíme rozšířit o výpočet tohoto atributu a generování odpovídajících konverzí a aritmetických operátorů. Pro převod hodnoty `y` typu integer na reálnou hodnotu `x` budeme generovat

na vhodných místech instrukci `x := inttoreal y` a budeme rozlišovat typ prováděné aritmetické operace. Například pro pravidlo $E \to E_1 + E_2$ můžeme použít sémantickou akci z obr. 8.12.

V reálné implementaci výrazů se smíšenými operandy se nevytváří samostatné pravidlo pro každý operátor; spíše se používá společný podprogram, jehož jedním parametrem je operátor, pro který se má vygenerovat kód. Často se generování kódu pro výrazy také řeší pomocí tabulek — například můžeme použít tabulku, ze které pro zadané typy operandů získáme informaci o tom, zda je tato kombinace přípustná a zda generovat určitou typovou konverzi pro některý z operandů.

**Příklad 8.4.** Za předpokladu, že proměnné `x` a `y` mají typ *real* a `i` a `j` typ *integer*, můžeme pro vstup `x := y + i * j` vygenerovat kód

```
t1 := i int+ j
t3 := inttoreal t1
t2 := y real+ t3
x  := t2
```

■

## 8.4 Booleovské výrazy

Booleovské výrazy se v programovacích jazycích používají ke dvěma hlavním účelům — pro výpočet logických hodnot a (především) jako podmínky v příkazech pro změnu toku řízení jako je např. podmíněný příkaz nebo příkaz cyklu.

Booleovské výrazy jsou tvořené operátory jako **and**, **or** nebo **not** a operandy, kterými mohou být booleovské konstanty, proměnné nebo relační výrazy. V některých jazycích mohou být operandy booleovských výrazů hodnoty i jiných typů. Pro další výklad budeme vycházet z této gramatiky booleovského výrazu:

$$E \to E \textbf{ or } E \mid E \textbf{ and } E \mid (E) \mid \textbf{id relop id} \mid \textbf{true} \mid \textbf{false}$$

Terminální symbol **relop** bude mít atribut *op* určující jeden ze šesti relačních operátorů.

Pro reprezentaci booleovských hodnot a překlad booleovských výrazů se používají dvě základní metody. První metoda reprezentuje logické hodnoty jako čísla a vyhodnocuje booleovské výrazy stejně jako aritmetické. Pro kódování booleovských hodnot se často používá 0 jako false a 1 nebo nenulová hodnota jako true.

Druhou základní metodou je reprezentace booleovských výrazů tokem řízení, tj. pozicí dosaženou v programu. Tato metoda je zvláště výhodná pro implementaci řídicích příkazů, neboť umožňuje jejich efektivnější vyhodnocování — pokud např. ve výrazu $E_1$ **or** $E_2$ zjistíme, že hodnota jednoho operandu je true, nemusíme již vyhodnocovat druhý operand.

To, zda můžeme použít první nebo druhou metodu, je dáno sémantikou implementovaného programovacího jazyka. Dovoluje-li jazyk ponechat některé části výrazu nevyhodnocené, může překladač provést optimalizaci booleovského výrazu a vyhodnotit jen tu část výrazu, kterou potřebuje. Pokud však některý z operandů má vedlejší účinek (např. se v něm volá funkce, která mění hodnotu nějaké globální proměnné), můžeme při zkráceném vyhodnocení výrazu dostat neočekávané výsledky. Obecně nelze říci, která z obou metod je výhodnější; některé překladače umožňují pomocí parametrů metodu překladu určit nebo dovedou vybrat vhodnou metodu na základě analýzy každého konkrétního výrazu.



### 8.4.1 Reprezentace booleovských výrazů číselnou hodnotou

Za předpokladu, že budeme logické hodnoty reprezentovat čísly 0 a 1 a provádět úplné vyhodnocení, můžeme výraz `a or b and not c` přeložit do tříadresového kódu jako

```
t1 := not c
t2 := b and t1
t3 := a or t2
```

Pokud výraz obsahuje relační operátor, musíme z výsledku relace nejprve odvodit příslušnou číselnou hodnotu. Například výraz `x < y` se přeloží jako

```
100: if x < y goto 103
101: t1 := 0
102: goto 104
103: t1 := 1
```

Překladové schéma pro generování tříadresového kódu pro booleovské výrazy je na obr. 8.13. Předpokládáme, že procedura *emit* zapisuje do výstupního souboru tříadresové příkazy a že proměnná *neststat* obsahuje index následujícího příkazu tříadresového kódu, zvyšovaný procedurou *emit*.

| | |
|---|---|
| $E \to E_1$ **or** $E_2$ | { *E.place* := *newtemp*;<br>*emit*(*E.place* `':='` $E_1$*.place* `'or'` $E_2$*.place*) } |
| $E \to E_1$ **and** $E_2$ | { *E.place* := *newtemp*;<br>*emit*(*E.place* `':='` $E_1$*.place* `'and'` $E_2$*.place*) } |
| $E \to$ **not** $E_1$ | { *E.place* := *newtemp*;<br>*emit*(*E.place* `':='` `'not'` $E_1$*.place*) } |
| $E \to ( E_1 )$ | { *E.place* := $E_1$*.place*} |
| $E \to$ **id**$_1$ **relop id**$_2$ | { *E.place* := *newtemp*;<br>*emit*(`'if'` **id**$_1$*.place* **relop**.*op* **id**$_2$*.place* `'goto'` *nextstat* + 3);<br>*emit*(*E.place* `':='` `'0'`);<br>*emit*(`'goto'` *nextstat* + 2);<br>*emit*(*E.place* `':='` `'1'`) } |
| $E \to$ **true** | { *E.place* := *newtemp*;<br>*emit*(*E.place* `':='` `'1'`) } |
| $E \to$ **false** | { *E.place* := *newtemp*;<br>*emit*(*E.place* `':='` `'0'`) } |

Obrázek 8.13: Překladové schéma pro číselnou reprezentaci booleovských výrazů

**Příklad 8.5.** Na základě schematu z obr. 8.13 můžeme pro booleovský výraz `a < b or c < d and e < f` vygenerovat kód uvedený na obr. 8.14. ■

### 8.4.2 Zkrácené vyhodnocování booleovských výrazů

Metoda zkráceného vyhodnocování booleovských výrazů umožňuje zpracovat booleovské výrazy bez jejich úplného vyhodnocení. Při této metodě se logické hodnoty nereprezentují jako data; každé kombinaci logických hodnot operandů ve výrazu místo toho odpovídá určitá pozice ve vygenerovaném kódu, na kterou se program dostane pomocí podmíněných a nepodmíněných skoků. Například na obr. 8.14 můžeme hodnotu proměnné `t1` odvodit z toho,

```
100: if a < b goto 103        107: t2 := 1
101: t1 := 0                  108: if e < f goto 111
102: goto 104                 109: t3 := 0
103: t1 := 1                  110: goto 112
104: if c < d goto 107        111: t3 := 1
105: t2 := 0                  112: t4 := t2 and t3
106: goto 108                 113: t5 := t1 or t4
```

Obrázek 8.14: Překlad výrazu `a < b or c < d and e < f`

zda se dostaneme na řádek 101 nebo 103 a podle toho pokračovat ve vyhodnocování zbytku výrazu; samotná hodnota proměnné `t1` je redundantní.

Překladové schéma na obr. 8.15 využívá pro zkrácené vyhodnocení výrazu dědičných atributů $E.true$ a $E.false$, které obsahují jméno návěští, na které má program přejít, je-li hodnota příslušného podvýrazu true, resp. false. Nonterminál $M$ s atributem $M.lab$ slouží pouze pro vygenerování návěští před vyhodnocením druhého operandu binárního operátoru. Funkce *newlabel* při každém volání vrátí nové, ještě nepoužité návěští.

| | |
|---|---|
| $E \to E_1$ **or** $M\ E_2$ | $\{\ E_1.true := E_2.true := E.true;$<br>$E_1.false := M.lab := newlabel;$<br>$E_2.false := E.false\ \}$ |
| $E \to E_1$ **and** $M\ E_2$ | $\{\ E_1.false := E_2.false := E.false;$<br>$E_1.true := M.lab := newlabel;$<br>$E_2.true := E.true\ \}$ |
| $E \to$ **not** $E_1$ | $\{\ E_1.true := E.false;$<br>$E_2.false := E.true\ \}$ |
| $E \to (\ E_1\ )$ | $\{\ E_1.true := E.true;$<br>$E_1.false := E.false\ \}$ |
| $E \to$ **id**$_1$ **relop id**$_2$ | $\{$ *gen*('`if`' **id**$_1$.*place* **relop**.*op* **id**$_2$.*place* '`goto`' $E.t$);<br>*gen*('`goto`' $E.false$) $\}$ |
| $E \to$ **true** | $\{$ *gen*('`goto`' $E.true$) $\}$ |
| $E \to$ **false** | $\{$ *gen*('`goto`' $E.false$) $\}$ |
| $M \to \epsilon$ | $\{$ *gen*($M.lab$ '`:`') $\}$ |

Obrázek 8.15: Překladové schéma pro zkrácené vyhodnocení booleovských výrazů

Pro každý relační operátor se vygeneruje podmíněný skok na návěští $E.true$ a nepodmíněný skok na návěští $E.false$. Všechny další operace spočívají pouze ve vhodném vytváření a kombinování návěští pro tyto dva skoky. Například předpokládejme, že máme výraz ve tvaru $E_1$ **and** $E_2$. Má-li $E_1$ hodnotu false, bude mít i celý výraz $E$ hodnotu false a můžeme tedy pro $E_1.false$ použít hodnotu $E.false$. Má-li $E_1$ hodnotu true, musíme vyhodnotit ještě $E_2$, takže použijeme $E_1.true$ jako návěští prvního příkazu pro $E_2$. Pro vyhodnocení $E_2$ pak již můžeme použít stejná návěští jako pro celý výraz $E$. Podobná úvaha platí i pro operátor **or**. Pro výraz ve tvaru **not** $E$ nepotřebujeme dokonce vůbec žádný kód, pouze vyměníme úlohy atributů $E.true$ a $E.false$.

**Příklad 8.6.** Na základě schematu z obr. 8.15 můžeme pro booleovský výraz `a < b or c < d and e < f` vygenerovat kód uvedený na obr. 8.16. ■



```
    if a < b goto Ltrue
    goto L1
L1: if c < d goto L2
    goto Lfalse
L2: if e < f goto Ltrue
    goto Lfalse
```

Obrázek 8.16: Zkrácený překlad výrazu `a < b or c < d and e < f`

Kód vygenerovaný podle schématu z obr. 8.15 není optimální. Například na obr. 8.16 lze vypustit druhý řádek, aniž se nějak ovlivní funkce programu. Vygenerovaný kód lze optimalizovat buď dodatečně, nebo je možné do překladového schematu začlenit akce, které budou určité optimalizace provádět již během generování. Často lze kód například vylepšit obrácením testované podmínky, např. přepíšeme-li třetí řádek na obr. 8.16 do tvaru

```
L1: if c >= d goto Lfalse
```

můžeme vypustit i následující skok na návěští `Lfalse`.

V některých jazycích, jako je např. jazyk C, se booleovské výrazy mohou libovolně kombinovat s ostatními aritmetickými výrazy. Schéma pro překlad takových kombinovaných výrazů musí zajistit přechod mezi reprezentací tokem řízení a číselnou reprezentací logických hodnot. Obr. 8.17 ukazuje dvě taková pravidla pro aritmetický výraz $AE$ a booleovský výraz $BE$.

| | |
|---|---|
| $AE \to BE$ | $\{\ BE.true := newlabel;$<br>$BE.false := newlabel;$<br>$templab := newlabel;$<br>$AE.place := newtemp;$<br>*gen*($BE.true$ '`:`');<br>*gen*($AE.place$ '`:=`' '`1`');<br>*gen*('`goto`' *templab*);<br>*gen*($BE.false$ '`:`');<br>*gen*($AE.place$ '`:=`' '`0`');<br>*gen*(*templab* '`:`') $\}$ |
| $BE \to AE$ | $\{$ *gen*('`if`' $AE.place$ '`<>`' '`0`' '`goto`' $BE.true$);<br>*gen*('`goto`' $BE.false$)$\}$ |

Obrázek 8.17: Pravidla pro překlad smíšených booleovských výrazů

## 8.5 Příkazy pro změnu toku řízení

Nyní se pokusíme předvedené metody překladu booleovských výrazů začlenit do překladu řídicích příkazů. Budeme uvažovat příkazy generované následující gramatikou:

$$S \to \textbf{if } E \textbf{ then } S_1 \mid \textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2 \mid \textbf{while } E \textbf{ do } S_1$$

V těchto pravidlech je vždy $E$ booleovský výraz. Při překladu s úplným vyhodnocením bude mít $E$ syntetizovaný atribut $E.place$, jméno proměnné obsahující hodnotu výrazu, při zkrá-

ceném překladu tokem řízení bude mít $E$ naopak dva *dědičné* atributy obsahující návěští pro hodnotu true ($E.true$) a false ($E.false$), stejně jako v předcházejících odstavcích.

| | |
|---|---|
| $S \rightarrow$ **if** $E$ **then** $M\ S_1$ | $\{\ E.true := newlabel;$ <br> $E.false := newlabel;$ <br> $M.lab := E.true;$ <br> $gen(E.false\ \text{':'})\ \}$ |
| $S \rightarrow$ **if** $E$ **then** $M\ S_1$ **else** $N\ M\ S_2$ | $\{\ E.true := newlabel;$ <br> $E.false := newlabel;$ <br> $M_1.lab := E.true;$ <br> $M_2.lab := E.false;$ <br> $N.lab := newlabel;$ <br> $gen(N.lab\ \text{':'})\ \}$ |
| $S \rightarrow$ **while** $M_1\ E$ **do** $M_2\ S_1$ | $\{\ M_1.lab := newlabel;$ <br> $M_2.lab := E.true;$ <br> $gen(\text{'goto'}\ M_1.lab);$ <br> $gen(E.false\ \text{':'})\ \}$ |
| $M \rightarrow \epsilon$ | $\{\ gen(M.lab\ \text{':'})\ \}$ |
| $N \rightarrow \epsilon$ | $\{\ gen(\text{'goto'}\ N.lab)\ \}$ |

Obrázek 8.18: Překladové schéma pro řídicí příkazy

Překladové schéma na obr. 8.18 vychází z předpokladu, že booleovské výrazy se překládají zkráceně. Pro vkládání návěští a skoků do řídicích konstrukcí se zde používají nonterminály $M$ a $N$ s dědičným atributem *lab* označujícím jméno návěští pro definici nebo skok. Nonterminál $N$ zajišťuje v úplném příkazu **if** přeskok části za **else** při splnění podmínky.

**Příklad 8.7.** Uvažujme příkaz

```
while a < b do
   if c < d then
      x := y + z
   else
      x := y - z
```

Na základě schémat z obr. 8.10, 8.15 a 8.18 můžeme pro tento příkaz vygenerovat následující tříadresový kód:

```
L1:  if a < b goto L2
     goto Lnext
L2:  if c < d goto L3
     goto L4
L3:  t1 := y + z
     x := t1
     goto L1
L4:  t2 := y - z
     x := t2
     goto L1
Lnext:
```

Pokud obrátíme směr relací v prvním a třetím řádku, můžeme vypustit za nimi následující nepodmíněné skoky. ■



## 8.6 Selektivní příkazy

Selektivní příkazy typu switch nebo case jsou dostupné v mnoha jazycích. Představují vlastně zobecněný příkaz if s více variantami. Obecně mají tyto příkazy strukturu obdobnou jako na obr. 8.19. Výraz $E$ v záhlaví selektivního příkazu se vyhodnotí a v případě, že se jeho hodnota rovná některé z uvedených konstant $V_i$, provede se příkaz $S_i$ uvedený za konstantou. V opačném případě, pokud je uvedena varianta **default**, se provede implicitní příkaz $S_{def}$.

**switch** $E$
  **begin**
    **case** $V_1$ : $S_1$
    **case** $V_2$ : $S_2$
    ...
    **case** $V_n$ : $S_n$
    **default** : $S_{def}$
  **end**

Obrázek 8.19: Struktura selektivního příkazu

Selektivní příkaz můžeme implementovat mnoha různými způsoby v závislosti na intervalu, ve kterém leží uvedené konstanty, velikosti mezer mezi konstantami (počtu nevyužitých hodnot uvnitř intervalu mezi největší a nejmenší konstantou) a preferovaných vlastnostech vygenerovaného kódu (optimalizace na čas nebo velikost kódu). Obecně se používají následující metody:

- *Posloupnost podmíněných skoků.* Tato nejjednodušší varianta je výhodná pouze při malém počtu položek; každý podmíněný skok testuje jednu uvedenou variantu.
- *Tabulka dvojic.* Vytvoříme vyhledávací tabulku obsahující vždy hodnotu konstanty a návěští začátku příkazu pro tuto variantu. V tabulce pak můžeme vyhledávat některou z běžných metod. Není-li hodnota výrazu v tabulce nalezena, provede se implicitní varianta. Pro velké počty návěští se někdy používá tabulek s rozptýlenými položkami.
- *Rozskoková tabulka.* V případě, že hodnoty konstant dostatečně hustě zaplňují určitý interval hodnot, například $< i_{min}, i_{max} >$, můžeme vytvořit pole návěští obsahující v $j$-tém prvku návěští pro hodnotu $i_{min} + j$. Při vyhodnocení selektivního příkazu se nejprve zjistí, zda hodnota výrazu $e$ leží v intervalu $< i_{min}, i_{max} >$ a pokud ano, provede se nepřímý skok na návěští uložené v $(e - i_{min})$-té položce tabulky. Nevyužité pozice v tabulce se zaplní návěštím pro implicitní variantu nebo chybu.

V praxi se rovněž používají modifikace uvedených metod nebo jejich kombinace. Například můžeme kromě podmíněného skoku testujícího rovnost hodnoty výrazu s některou konstantou vygenerovat zároveň odskok, je-li hodnota menší, a dále pak testovat odděleně již menší podmnožiny hodnot (jde vlastně o implementaci binárního vyhledávacího stromu pomocí toku řízení). Po vymezení dostatečně kompaktní podmnožiny hodnot pak můžeme pro konečné vyhodnocení použít rozskokové tabulky.

Překlad selektivního příkazu do tříadresového kódu má strukturu jako na obr. 8.20. Po vyhodnocení výrazu $E$ se provede odskok na vlastní tělo selektivního příkazu, čímž se přeskočí

kód vygenerovaný pro jednotlivé varianty. Každá varianta je pak označena návěštím a končí skokem na příkaz následující za selektivním příkazem (V jazyce C se tento skok generuje pouze pro explicitně zapsaný příkaz `break`).

```
      kód pro vyhodnocení E do proměnné t
      goto test
L1:   kód pro S1
      goto next
L2:   kód pro S2
      goto next
        ...
Ln:   kód pro Sn
      goto next
test:if t = V1 goto L1
      if t = V2 goto L2
        ...
      if t = Vn goto Ln
next:
```

Obrázek 8.20: Překlad selektivního příkazu

Část kódu uvedená mezi návěštími `test` a `next` odpovídá vlastnímu selektivnímu příkazu a její struktura tedy závisí na zvolené metodě překladu. Pokud nechceme tento problém řešit již při generování mezikódu, lze rozšířit mezikód o speciální příkazy reprezentující selektivní příkaz, např. takto:

```
test: select t,Ldef
        case   V1,L1
        case   V2,L2
        ...
        case   Vn,Ln
next:
```

Příkaz `select` má jako parametr odkaz na místo, kde je uložena hodnota vypočteného výrazu a návěští pro implicitní variantu, příkaz `case` definuje hodnotu konstanty a návěští příslušného příkazu pro každou uvedenou variantu. Konec seznamu příkazů `case` je jednoduše rozpoznatelný podle návěští, které musí vždy následovat. O skutečné metodě překladu se pak může rozhodnout až při generování kódu, kdy lze například využít některých speciálních instrukcí procesoru.

## 8.7 Backpatching

V předchozích odstavcích jsme si předvedli několik různých metod generování mezikódu pro řídicí příkazy. Nezabývali jsme se však vlastním přiřazením adres pro jednotlivá návěští, obcházeli jsme jej použitím symbolických jmen. Při víceprůchodovém překladu se toto přiřazení může provést v samostatném průchodu vygenerovaným kódem, kdy už jsou známy všechny vygenerované instrukce i rozmístění jednotlivých návěští. Pokud se ale překlad provádí jednoprůchodově, dostáváme se velmi často do situace, kdy neznáme v určitém bodě cílovou adresu návěští, neboť kód, který bude tímto návěštím označen, ještě nebyl vygenerován.



Tento problém můžeme řešit tak, že necháme adresu návěští prázdnou a udržujeme seznam adres, ze kterých se na každé takové návěští odkazujeme. V okamžiku, kdy dosáhneme místa obsahujícího definici návěští, zpětně jeho adresu doplníme do všech míst uvedených v seznamu. Tato metoda zpětných oprav se nazývá *backpatching*. Implementaci si ukážeme na jednoprůchodovém překladu řídicích příkazů s logickými výrazy vyhodnocovanými zkráceně.

Budeme předpokládat, že generujeme čtveřice uložené v poli; návěští pak budou indexy do pole čtveřic. Adresa následující volné čtveřice bude uložena v proměnné $nextquad$. Pro manipulaci se seznamy návěští budeme používat následující podprogramy:

1. $makelist(i)$ vytvoří nový seznam obsahující pouze index $i$; vrátí ukazatel na vytvořený seznam.
2. $merge(p_1, p_2)$ spojí dva seznamy, na které ukazuje $p_1$ a $p_2$ do jediného a vrátí ukazatel na takto vytvořený nový seznam.
3. $backpatch(p, i)$ vloží $i$ jako adresu návěští do všech příkazů obsažených v seznamu, na který ukazuje $p$.

### 8.7.1 Booleovské výrazy

Při překladu booleovských výrazů doplníme do gramatiky nonterminál (marker) $M$, který bude sloužit pro uschování současné hodnoty čítače čtveřic $nextquad$. Upravená gramatika, pro kterou budeme dále vytvářet překladové schéma, je následující:

| | |
|---|---|
| (1) | $E \to E_1$ **or** $M\ E_2$ |
| (2) | $\mid E_1$ **and** $M\ E_2$ |
| (3) | $\mid$ **not** $E_1$ |
| (4) | $\mid ( E_1 )$ |
| (5) | $\mid$ **id**$_1$ **relop** **id**$_2$ |
| (6) | $\mid$ **true** |
| (7) | $\mid$ **false** |
| (8) | $M \to \epsilon$ |

Nonterminál $E$ používá syntetizované atributy $truelist$ a $falselist$, obsahující seznamy neúplných instrukcí s odskoky při hodnotě výrazu true a false. Sémantické akce vycházejí z následující úvahy: Například je-li v pravidle $E \to E_1$ **and** $M\ E_2$ hodnota $E_1$ false, je i hodnota $E$ false, takže všechny příkazy v $E_1.falselist$ se stanou částí $E.falselist$. Je-li však $E_1$ true, musíme nejprve testovat $E_2$, takže cílovou adresou pro instrukce v $E_1.truelist$ musí být adresa začátku kódu pro vyhodnocení $E_2$. Tu získáme pomocí markeru $M$, jehož atribut $M.quad$ obsahuje právě index prvního příkazu pro $E_2$. S pravidlem $M \to \epsilon$ je svázána sémantická akce

$$\{\ M.quad := nextquad\ \}$$

. Hodnota čítače $nextquad$ uschovaná tímto nonterminálem bude sloužit pro zpětné opravy adres v seznamu $E_1.truelist$ v okamžiku, kdy dosáhneme konce pravidla pro operátor **and**. Celé překladové schéma pro booleovské výrazy je uvedeno na obr. 8.21.
Sémantická akce (5) generuje dva skoky, podmíněný a nepodmíněný. Oba tyto skoky směřují na dosud neznámou adresu, proto se jejich adresy zařadí do seznamů $E.truelist$ a $E.falselist$.
V pravidlech (6) a (7) se reprezentuje konstanta **true** a **false** jako nepodmíněný skok, jehož adresa se opět zařadí do příslušného seznamu neúplných skoků; druhý seznam v tomto případě

| | | |
|---|---|---|
| (1) | $E \to E_1$ **or** $M\ E_2$ | { $backpatch(E_1.falselist, M.quad)$; <br> $E.truelist := merge(E_1.truelist, E_2.truelist)$; <br> $E.falselist := E_2.falselist$ } |
| (2) | $E \to E_1$ **and** $M\ E_2$ | { $backpatch(E_1.truelist, M.quad)$; <br> $E.truelist := E_2.truelist$; <br> $E.falselist := merge(E_1.falselist, E_2.falselist)$ } |
| (3) | $E \to$ **not** $E_1$ | { $E.truelist := E_1.falselist$; <br> $E.falselist := E_1.truelist$ } |
| (4) | $E \to (\ E_1\ )$ | { $E.truelist := E_1.truelist$; <br> $E.falselist := E_1.falselist$ } |
| (5) | $E \to$ **id**$_1$ **relop id**$_2$ | { $E.truelist := makelist(nextquad)$; <br> $E.falselist := makelist(nextquad + 1)$; <br> $emit($`'if'` **id**$_1.place$ **relop**$.op$ **id**$_2.place$ `'goto _'`$)$; <br> $emit($`goto _`$)$ } |
| (6) | $E \to$ **true** | { $E.truelist := makelist(nextquad)$; <br> $E.falselist :=$ **nil**; <br> $emit($`'goto _'`$)$ } |
| (7) | $E \to$ **false** | { $E.falselist := makelist(nextquad)$; <br> $E.truelist :=$ **nil**; <br> $emit($`'goto _'`$)$ } |
| (8) | $M \to \epsilon$ | { $M.quad := nextquad$ } |

Obrázek 8.21: Překladové schéma pro booleovské výrazy s backpatchingem

bude prázdný. Uvedené schéma může být použito přímo při překladu zdola nahoru, neboť všechny sémantické akce se vyskytují na konci pravidel a mohou se tedy provádět zároveň s redukcemi.

**Příklad 8.8.** Uvažujme opět výraz `a<b or c<d and e<f` jako v příkladu 8.6. Odpovídající ohodnocený derivační strom je uveden na obr. 8.22 (názvy atributů jsou zde zkráceny). Pro podvýraz `a<b` se na základě pravidla (5) vygenerují dvě čtveřice

```
100: if a < b goto _
101: goto _
```

Nonterminál $M$ v pravidle (1) zaznamená hodnotu $nextquad$, která je nyní 102. Redukce `c<d` na $E$ podle pravidla (5) vygeneruje čtveřice

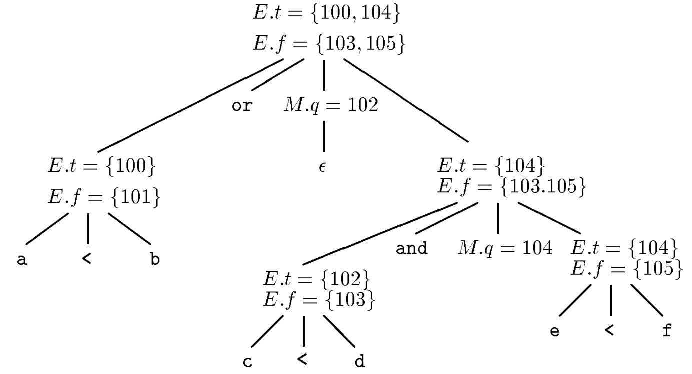
```
102: if c < d goto _
103: goto _
```

Nyní máme k dispozici nonterminál $E_1$ z pravidla (2). Následující marker $M$ zaznamená současnou hodnotu $nextquad$, tj. 104. Redukce `e<f` na $E$ opět podle pravidla (5) vygeneruje

```
104: if y < f goto _
105: goto _
```

Dále nastane redukce podle pravidla $E \to E_1$ **and** $M\ E_2$. Odpovídající sémantická akce volá proceduru $backpatch(\{102\}, 104)$, která doplní adresu 104 do příkazu 102. Konečně redukce podle pravidla $E \to E_1$ **or** $M\ E_2$ volá $backpatch(\{101\}, 102)$, která doplní adresu 102 do příkazu 101. Tím dostaneme konečnou posloupnost příkazů ve tvaru






Obrázek 8.22: Ohodnocený derivační strom pro `a<b or c<d and e<f`

```
100: if a < b goto _
101: goto 102
102: if c < d goto 104
103: goto _
104: if e < f goto _
105: goto _
```

přičemž pro nonterminál $E$ reprezentující celý výraz budeme mít atributy $E.falselist = \{103, 105\}$ a $E.truelist = \{100, 104\}$. To znamená, že celý výraz má hodnotu true, provedou-li se skoky v příkazech 100 nebo 104, a hodnotu false, provedou-li se skoky v příkazech 103 nebo 105. Cílové adresy těchto skoků se doplní dále během překladu v závislosti na tom, co se má udělat při které hodnotě výrazu. ■

### 8.7.2 Překlad řídicích příkazů

Nyní předvedeme, jakým způsobem se dá backpatching použít při jednoprůchodovém překladu řídicích příkazů. Pro překlad budeme používat následující gramatiku:

| | |
|---|---|
| (1) | $S \to$ **if** $E$ **then** $S$ |
| (2) | $\quad\mid$ **if** $E$ **then** $S$ **else** $S$ |
| (3) | $\quad\mid$ **while** $E$ **do** $S$ |
| (4) | $\quad\mid$ **begin** $L$ **end** |
| (5) | $\quad\mid A$ |
| (6) | $L \to L\ ;\ S$ |
| (7) | $\quad\mid S$ |

Nonterminál $S$ označuje příkaz, $L$ seznam příkazů, $A$ přiřazovací příkaz a $E$ booleovský výraz. Pro booleovský výraz použijeme výsledků předchozího odstavce, struktura přiřazovacího příkazu, případně dalších jednoduchých příkazů nás nyní nebude zajímat.

Pro překlad zvolíme opět ten přístup, že budeme zpětně doplňovat adresy skoků v tom okamžiku, kdy dosáhneme jejich cílového příkazu. Kromě dvou seznamů adres pro booleovské

výrazy budeme rovněž potřebovat seznam adres skoků na kód, který následuje za příkazy generovanými neterminály $S$ a $L$; na tento seznam bude ukazovat atribut *nextlist*.

Při překladu příkazu $S \rightarrow$ **while** $E$ **do** $S_1$ budeme potřebovat dvě návěští: jedno pro označení začátku celého příkazu $S$ a jedno pro tělo cyklu $S_1$. Adresy těchto návěští opět zaznamenáme pomocí dvou výskytů markeru $M$ na příslušných pozicích v pravidle:

$$S \rightarrow \textbf{while}\ M_1\ E\ \textbf{do}\ M_2\ S_1$$

S neterminálem $M$ bude opět svázáno pravidlo, které do atributu $M.quad$ uloží hodnotu čítače *nextquad*. Po zpracování těla $S_1$ se řízení vrací na začátek cyklu, takže při redukci **while** $M_1$ $E$ **do** $M_2$ $S_1$ na $S$ přepíšeme cílové adresy skoků ze seznamu $S_1.nextlist$ na $M_1.quad$. Vzhledem k tomu, že posledním příkazem $S_1$ nemusí být skok, musíme rovněž za tělo $S_1$ doplnit explicitní skok na začátek kódu pro $E$. Nakonec do příkazů v seznamu $E.truelist$ doplníme adresu začátku $S_1$, tj. $M_2.quad$ a $E.falselist$ se stane hodnotou atributu $S.nextlist$ celého příkazu $S$.

Překlad příkazu **if** $E$ **then** $S_1$ **else** $S_2$ je ještě poněkud složitější, neboť potřebujeme za kód vygenerovaný z $S_1$ vložit skok přes kód pro $S_2$. K tomu využijeme dalšího markeru $N$ s atributem $N.nextlist$, seznamem obsahujícím pouze adresu čtveřice s příkazem `goto _`, který vygeneruje sémantická akce spojená s $N$. Úplné překladové schéma pro řídicí příkazy je uvedeno na obr. 8.23.

(1) $S \rightarrow$ **if** $E$ **then** $M_1$ $S_1$ $N$ **else** $M_2$ $S_2$
  { $backpatch(E.truelist, M_1.quad)$;
  $backpatch(E.falselist, M_2.quad)$;
  $S.nextlist := merge(S_1.nextlist, merge(N.nextlist, S_2.nextlist))$ }

(2) $N \rightarrow \epsilon$  { $N.nextlist := makelist(nextquad)$;
  *emit*('`goto _`') }

(3) $M \rightarrow \epsilon$  { $M.quad := nextquad$ }

(4) $S \rightarrow$ **if** $E$ **then** $M$ $S_1$
  { $backpatch(E.truelist, M.quad)$;
  $S.nextlist := merge(E.falselist, S_1.nextlist)$ }

(5) $S \rightarrow$ **while** $M_1$ $E$ **do** $M_2$ $S_1$
  { $backpatch(S_1.nextlist, M_1.quad)$;
  $backpatch(E.truelist, M_2.quad)$;
  $S.nextlist := E.falselist$;
  *emit*('`goto`' $M_1.quad$) }

(6) $S \rightarrow$ **begin** $L$ **end**  { $S.nextlist := L.nextlist$ }

(7) $S \rightarrow A$  { $S.nextlist :=$ **nil** }

(8) $L \rightarrow L_1$ ; $M$ $S$  { $backpatch(L_1.nextlist, M.quad)$;
  $L.nextlist := S.nextlist$ }

(9) $L \rightarrow S$  { $L.nextlist := S.nextlist$ }

Obrázek 8.23: Překladové schéma pro řídicí příkazy s backpatchingem

Při zpracování návěští a příkazů skoku jako součástí zdrojového jazyka je třeba provádět navíc určité kontroly, například zda existuje právě jedna definice návěští, nebo zda cílová adresa skoku nemíří dovnitř složené konstrukce. Některé jazyky, jako např. Pascal, dále vyžadují, aby všechna návěští byla předem deklarována.



Pokud překladač rozpozná příkaz skoku, např. `goto L`, musí zkontrolovat, zda je v rozsahu platnosti návěští `L` právě jedna jeho definice. Pokud se návěští ještě nevyskytlo, uloží se do tabulky symbolů a přiřadí se mu jako atribut seznam tvořený adresou vygenerované čtveřice s příkazem skoku. Při všech dalších výskytech návěští v příkazu skoku ještě před jeho definicí se do tohoto seznamu pouze přidávají adresy odpovídajících skokových instrukcí. V okamžiku, kdy se vyskytne definice takto již použitého návěští, zavolá se procedura *backpatch*, které se předá seznam nedokončených skokových instrukcí a současná hodnota *nextquad*. Hodnota *nextquad* se rovněž uloží do tabulky symbolů jako skutečná adresa návěští, která se pak může v následujících příkazech skoku použít přímo.

### 8.7.3 Volání podprogramů

Podprogramy jsou tak důležité a často používané programové konstrukce, že je nutné, aby pro volání a návraty z podprogramů generoval překladač co nejefektivnější kód. V některých případech se mohou akce spojené s předáváním řízení mezi podprogramy provádět ve spolupráci se systémem řízení běhu programu.

Jak již bylo uvedeno v kapitole 6, volání podprogramu je obvykle standardizované ve tvaru určité volací posloupnosti. I když se volací posloupnosti od sebe liší i pro jednotlivé implementace jednoho programovacího jazyka, jsou obvykle tvořeny následujícími akcemi:

Je-li zavolán podprogram, musí se přidělit prostor pro alokační záznam volaného podprogramu. Musí se vyhodnotit předávané argumenty a dát je k dispozici volanému podprogramu na určitém známém místě. Dále je třeba upravit vazební ukazatele tak, aby měl volaný podprogram zajištěn přístup ke správným datům v nadřazených blocích. Stav volajícího podprogramu musí být uložen tak, aby mohl být znovu obnoven při návratu. Na známé místo se také uloží návratová adresa, místo, na které se vrátí řízení po ukončení volaného podprogramu. Návratová adresa je obvykle místo následující za příkazem volání. Nakonec se musí vygenerovat skok na začátek volaného podprogramu.

Při návratu z podprogramu se rovněž musejí provést určité pevně stanovené akce. Je-li podprogram funkcí, je třeba uložit na známé místo výsledek. Dále se musí obnovit aktivační záznam volajícího podprogramu a provést skok na návratovou adresu.

Mezi úkoly volajícího a volaného podprogramu není žádná přesná hranice. Často se tyto úkoly rozdělují na základě vlastností zdrojového jazyka, cílového počítače a požadavků operačního systému.

# Literatura


[1] Adobe Systems, Inc.: *PostScript language reference manual.* Addison-Wesley, sixteenth edition, 1990.

[2] A. V. Aho, R. Sethi, J. D. Ullman: *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1987.

[3] J. P. Bennett: *Introduction to compiling techniques: a first course using ANSI C, LEX and YACC.* McGraw-Hill, 1990.

[4] A. I. Holub: *Compiler Design in C.* Prentice Hall, 1990.

[5] J. M. Honzí k: *Programovací techniky.* Učební texty vysokých škol. Ediční středisko VUT Brno, 1985.

[6] T. Hru ˇ ska. *Modelování sémantiky programovacích jazyků a využití modelů při implementaci překladačů.* Disertační práce, VUT Brno, 1983.

[7] D. E. Knuth: *The METAFONTbook.* Addison-Wesley, 1986.

[8] D. E. Knuth: *The TEXbook.* Addison-Wesley, 1987.

[9] L. Lamport: *LATEX: A Document Preparation System.* Addison-Wesley, 1986.

[10] B. Melichar: Bezkontextové, překladové a atributové gramatiky. In *Sborník konference MOP'85, 1. díl*, strany 23–116, 1985.

[11] Jean-Paul Tremblay, P. G. Sorenson: *The Theory and Practice of Compiler Writing.* Computer Science Series. McGraw-Hill, 1985.

[12] M. Češka a kol.: *Gramatiky a jazyky.* Skriptum VUT Brno. Ediční středisko VUT Brno, 1985.

[13] M. Češka, T. Hruška: *Gramatiky a jazyky — cvičení.* Skriptum VUT Brno. Ediční středisko VUT Brno, 1986.

[14] W. M. Waite, G. Goos: *Compiler construction.* Text and monographs in computer science. Springer-Verlag, 1984.