Pokroky matematiky, fyziky a astronomie

Jiří Kopřiva

Automatizace programování

Pokroky matematiky, fyziky a astronomie, Vol. 9 (1964), No. 3, 156--171

Persistent URL: http://dml.cz/dmlcz/137026

Terms of use:

© Jednota českých matematiků a fyziků, 1964

Institute of Mathematics of the Academy of Sciences of the Czech Republic provides access to digitized documents strictly for personal use. Each copy of any part of this document must contain these *Terms of use*.

This paper has been digitized, optimized for electronic delivery and stamped with digital signature within the project *DML-CZ: The Czech Digital Mathematics Library* http://project.dml.cz

Třídy ekvivalentních posloupností nazýváme distribuce.

Všechny dosud známé definice distribucí vedou ke konstrukci nových matematických objektů, zobecněných funkcí, které je možno bez omezení „derivovat".

Teorie distribucí aplikována na klasické integrální transformace, např. Fourierovu, vede k novým výsledkům a umožňuje studovat zobecněná řešení některých diferenciálních rovnic, která mají přímý význam v aplikacích matematiky.

Literatura

[1] J. HADAMARD: *Le problème de Cauchy et les équations aux dérivées partielles linéaires hyperboliques*. Paris 1932.

[2] M. RIESZ: L'intégrale de Riemann-Liouville et le problème de Cauchy. Acta Math. *81* (1949), 1.

[3] S. L. SOBOLEV: Methode nouvelle à resoudre le problème de Cauchy pour les équations linéaires hyperboliques normales. Matemat. sb., *1* (43) (1936), 39.

[4] S. L. SOBOLEV: *Někotoryje primenenija funkcionalnogo analiza v matematičeskoj fizike*. Leningrad 1950.

[5] L. SCHWARTZ: *Théorie des distributions* I, II. Paris 1950—51.

[6] P. A. M. DIRAC: *Principles of quantum mechanics*. Oxford-University Press.

[7] J. von NEUMANN: *Mathematische Grundlagen der Quantenmechanik*.

[8] I. M. GELFAND, G. E. ŠILOV: *Obobščennyje funkcii i dejstvija nad nimi*. Moskva 1958, vypusk 1—4.

AUTOMATIZACE PROGRAMOVÁNÍ

JIŘÍ KOPŘIVA, Brno

1. Z článku J. HOŘEJŠE [1] se mohl čtenář poučit o základních principech programování pro samočinné počítače. Při způsobu tam popsaném musí programátor vypracovat celý program výpočtu v *operačním kódu* počítače, na němž má být výpočet proveden. To znamená, že smí užít pouze těch operací, které počítač provádí na základě jednotlivých *instrukcí* svého operačního kódu. Výpočet musí být tedy zpravidla rozložen na elementární *aritmetické*, *logické* a *řadicí operace*.

Tento způsob programování je sice dosud nejběžnější, má však řadu nevýhod. Nehledě na to, že pro velmi rychlé počítače nemá tato metoda prakticky smysl, souvisí tyto nevýhody také s tím, že operační kód běžných samočinných počítačů je někdy velmi omezený. Obsahuje zpravidla instrukce pro elementární aritmetické operace, několik logických instrukcí (pro srovnávání čísel co do rovnosti či nerovnosti, logické násobení a sčítání) a několik řadicích instrukcí (pro porušení přirozené posloupnosti plnění instrukcí apod.). Např. cyklické výpočty, které se při iteračních i jiných numerických metodách velmi často vyskytují, musí být někdy dosti pracně sestavovány s použitím výše uvedených instrukcí. Při takovém způsobu programování

musí mít programátor přehled o tom, na která místa paměti ukládá nejen vstupní údaje a průběžné výsledky, ale i jednotlivé instrukce programu, popř. podprogramy.

Je zřejmé, že sestavení programu pro poněkud komplikovanější numerický výpočet je práce velmi zdlouhavá, mnohdy jednotvárná a vyčerpávající. Vyžaduje velikého soustředění, nemá-li se při ní udělat chyba, která pak znehodnotí celý výpočet. Hledání takových chyb je velice obtížné. Pravděpodobnost výskytu chyby v programu již uloženém do paměti počítače se zvyšuje tím, že program napsaný programátorem na předepsaných formulářích v příslušném kódu (pomocí číslic a někdy také písmen) přepisuje operátorka pomocí příslušného zařízení na děrnou pásku, která se pak vkládá do čtecího zařízení. Operátorka není zpravidla seznámena s detaily řešení úlohy. Tyto okolnosti jsou hlavní příčinou toho, že nepřiměřeně mnoho drahého času práce počítače se spotřebuje na zkoušení programu, tj. hledání případných chyb. Také čas, který matematik programátor ztráví nad přípravou programu, by jím mohl být lépe využit, a to produktivnějším způsobem na práci tvůrčího charakteru.

Závažný je i další nedostatek uvedeného způsobu programování. Řešení úlohy, které je zapsáno ve tvaru programu pro určitý samočinný počítač, je srozumitelné pouze tomu, kdo je obeznámen s operačním kódem tohoto počítače. Nehodí se tedy tento tvar zápisu řešení pro publikaci nebo sdělení jinému výpočtovému středisku, které pracuje s jiným samočinným počítačem.

Různé způsoby odstraňování, popř. zmenšování některých z těchto nedostatků a problémy, které s tím souvisí, zahrnujeme pod pojem *automatizace programování*. Je pro ni charakteristická snaha přenechat počítači jistou část práce programátora. Týká se to samozřejmě oné mechanické práce, jako je detailní rozepisování programu na jednotlivé instrukce, umísťování programu a dat v paměti apod. Jsou dva extrémní případy. Jeden záleží ve výše zmíněné nutnosti napsat program řešení úlohy ve tvaru posloupnosti elementárních aritmetických, logických a řadicích operací pomocí instrukcí operačního kódu počítače. Při opačném extrému by stačilo napsat řešení úlohy způsobem na první pohled srozumitelným každému matematikovi, tj. pomocí číslic, písmen a obvyklých znaků užívaných v matematice a formální logice.

V prvním případě je možné zahájit výpočet na počítači ihned po vložení programu do paměti počítače. V druhém případě je ovšem také nutné, aby před zahájením výpočtu měl počítač ve své paměti k dispozici opět detailně na jednotlivé instrukce rozepsaný program. Požaduje se však, aby toto rozepsání a umístění do paměti provedl počítač sám. Takový počítač musí tedy umět jednak přečíst program řešení, napsaný „matematickým" způsobem, jednak ho „přeložit" do svého jazyka, tj. zapsat ho do své paměti ve formě posloupnosti instrukcí svého operačního kódu. Toto překládání nazýváme obvykle *kompilací*. Kompilace může být buď provedena tzv. *kompilátorem*, tj. standardním programem uloženým předem do paměti, nebo může být schopnost kompilace vložena do počítače už při jeho stavbě (druhá možnost zatím pouze pokusně).

Je patrno, že extrémní případ „matematického" zápisu programu by vedl buď

k velmi složitému a dlouhému kompilátoru, který by zabíral velikou část paměti (pokud by se do ní vůbec vešel), nebo k příliš složitému, a tedy příliš drahému počítači. Ve skutečnosti se užívá různých mezistupňů mezi oběma extrémy. Čím blíže je tento mezistupeň matematickému zápisu, tím je snazší práce člověka programátora, ale tím jsou také vyšší požadavky na kompilátor nebo na počítač, a naopak. Na zajímavou souvislost upozorňuje žertem přední maďarský matematik prof. L. KALMÁR. Říká, že kolonialisté se neučili jazyku svých poddaných národů. Svá přání a rozkazy sdělovali dělníkům prostřednictvím několika vzdělanějších jedinců, které naučili potřebným slovům a frázím svého jazyka. Proč by se tedy měl člověk učit jazyku svého „poddaného", tj. stroje. Ať se stroj naučí rozumět jazyku člověka.

Mezi různé mezistupně automatizace programování patří využívání tzv. *standardních podprogramů* a *programování v symbolických adresách*. Tyto způsoby jsou ještě dosti blízké programování v operačním kódu samočinného počítače. Bližší k matematickému zápisu řešení jsou způsoby, kterým se někdy říká *algebraické*. Sem patří *operátorová metoda*, dosti rozšířená v Sovětském svazu (viz [2], [3]), a také užívání různých *autokódů* a umělých *programovacích jazyků*. Tyto jazyky a autokódy jsou sestavovány tak, aby bylo možné pomocí nich zapsat algoritmy řešení jistých tříd úloh, které mají být řešeny na samočinném počítači. Autokódy a některé programovací jazyky využívají vlastností určitého konkrétního počítače. Sem patří autokódy pro počítače ELLIOTT 803, LGP-30 a jiné, ale také např. umělý programovací jazyk FORTRAN, původně sestavený pro počítač IBM-704, později přizpůsobený počítačům IBM-705, IBM-709 atd. (viz [4]). V posledních letech dosáhl velikého rozšíření programovací jazyk ALGOL 60. Při jeho sestavování se autoři nevázali zvláštnostmi žádného konkrétního již fungujícího nebo konstruovaného počítače, ale pokusili se vytvořit jazyk, který by v sobě zahrnoval vlastnosti jazyků sestavených dříve pro určité počítače. Tento jazyk používá některých matematických a logických symbolů, ale i slov živého jazyka (angličtiny). Pro tyto své vlastnosti se zřejmě výborně hodí i k publikaci algoritmů pro řešení úloh různých typů a jejich sdělování mezi výpočetními středisky.

Ve druhé části článku naznačíme způsob sestavování programů v Algolu 60. Nakonec si ve třetí části povíme něco o problémech *matematické lingvistiky* v souvislosti s „gramatikou" tohoto jazyka.

Pro úplnost poznamenejme ještě, že se v poslední době v rámci snahy o zautomatizování programování uvažuje i o tzv. *bezadresových počítačích*. Byly dokonce postaveny prototypy takových počítačů, a to v Polsku a v USA.

2. V [5] se říká o *mezinárodním algoritmickém jazyku* ALGOL 60 (ALGOrithmic Language), že je to jazyk vhodný k takovému vyjádření velké třídy numerických výpočtů, které by se svou dostatečnou stručností hodilo pro automatické překládání do jazyka samočinného počítače. Jeho poslední varianta popsaná v [5] se dosti liší od původního tvaru popsaného v předběžné zprávě připravené na konferenci v Curychu v r. 1958. (Tento předběžný tvar je popsán ve [4]). Forma užívaná v současné době vzešla v podstatě z konference, která se konala ve dnech 11. až

16. ledna 1960 v Paříži. Sedm evropských účastníků (z Dánska, Anglie, Francie, záp. Německa, Holandska a Švýcarska) a šest účastníků ze Spojených států bylo vybráno na několika formálních i neformálních schůzkách, konaných předtím v Evropě i v Americe. Základem pro práci lednové konference byl jednak nový návrh zprávy, který vypracoval PETER NAUR na základě předběžné zprávy a doporučení přípravných schůzek, jednak připomínky, které byly posílány od různých zájemců do Algol Bulletinu v Evropě a do Communication of the ACM (Association of Computing Machinery) v USA, kde byly publikovány. U příležitosti symposia pořádaného na jaře r. 1962 v Římě organisací ICC (International Computation Centre) se konala 2. a 3. dubna schůzka některých autorů jazyka ALGOL 60. Na ní byly provedeny některé úpravy a vyjasnění zprávy a byla schválena zpráva [5].

Program výpočtu nebo obecněji nějakého postupu (algoritmu), jímž se má získat požadovaný výsledek, psaný v jazyku ALGOL 60, je posloupnost tzv. *příkazů*. Příkazem se v Algolu rozumí pokyn k provedení nějaké činnosti. Odpovídá tedy zhruba instrukci při programování v kódu počítače. Ve skutečnosti ovšem vyplnění příkazu v Algolu znamená zpravidla provedení celé řady elementárních operací, plněných na základě instrukcí kódu samočinného počítače. Příkazy programu se plní v tom pořadí, v jakém jsou napsány, není-li tato posloupnost předepsaným způsobem porušena (viz dále).

Jedním ze základních příkazů je *přiřazovací příkaz*. Jím se přiřazuje *proměnné* hodnota a má obvykle tvar

(1) *proměnná* : = *aritmetický výraz*

Proměnná je přitom obdobně jako při matematickém vyjádření zastupována svým jménem, kterému se zde říká *identifikátor* (proměnné). Může to být písmeno latinské abecedy (malé i velké), ale také libovolná posloupnost písmen a číslic, která však musí začínat písmenem. To poskytuje programátorovi možnost, aby si proměnné označoval takovými identifikátory, které mu připomínají jejich význam při výpočtu. Tak např. *a* nebo *A*, ale také *b* 19 (třeba pro b_{19} z přípravných zápisů programátora) nebo *Max* (pro největší prvek jistého souboru) jsou přípustné identifikátory proměnných.

Aritmetický výraz se tvoří obdobně jako v matematice tím, že proměnné a čísla spojujeme *aritmetickými operátory* tak, abychom dostali výraz, který má smysl, tj. kterému můžeme přiřadit obvyklým způsobem hodnotu, známe-li hodnoty všech proměnných v něm vystupujících. Při jeho sestavování můžeme nebo dokonce někdy musíme užívat kulatých *závorek*, abychom odstranili případné nejasnosti, např. kterou operaci provést dříve a kterou později.

Aritmetický výraz se vyhodnotí tím, že se užije tzv. *běžných hodnot* jednotlivých proměnných v něm vystupujících. Běžnou hodnotou proměnné se rozumí číselná hodnota, která je právě umístěna v té buňce paměti, jež je vyhrazena (dočasně nebo natrvalo pro celý výpočet) této proměnné.

Při vypisování čísel se namísto desetinné čárky užívá desetinné tečky. Nula před desetinnou tečkou se nemusí vypisovat, takže 0.324 se dá psát také .324. Před kladné číslo se může připojit znaménko +. Záporné číslo je předznačeno znaménkem −. Máme možnost užít tzv. *exponentové části*, která má tvar $_{10}$ *celé číslo* a má sama o sobě význam příslušné mocniny desítky. Napíšeme-li před ní číslo, má vzniklá konfigurace hodnotu předepsaného čísla násobeného příslušnou mocninou desítky. Můžeme tedy číslo − 31.004 zapsat v Algolu např. těmito způsoby: − 31.004, − .31004$_{10}$ 2, − 310040$_{10}$ −4, − 31004.00$_{10}$ − 3 atd. Za *celé číslo* se v Algolu považuje pouze posloupnost cifer, případně s předepsaným znaménkem. Tedy − 0772 je celé číslo, ale 112.0 nebo 2$_{10}$3 nikoliv, i když obě poslední jsou celá čísla v obvyklém smyslu.

Užívá se těchto aritmetických operátorů: + pro sčítání, − pro odčítání, × pro násobení, / pro dělení (někdy též ÷) a ↑ pro umocňování. Operátor × se nesmí nikdy vynechat. Psaní symbolů vedle sebe, které je pro vyznačení násobení obvyklé v matematice, znamená zde pouhé zřetězení, potřebné např. při vytváření identifikátorů. Uveďme si několik příkladů možného způsobu zápisu aritmetických výrazů v Algolu:

Výraz	Zápis v Algolu
$a \,.\, b \,.\, c$	$a \times b \times c$
$\dfrac{A^2 - B^2}{X + Y}$	$(A\uparrow 2 - B\uparrow 2)/(X + Y)$
$\cfrac{a}{1 + \cfrac{a^3}{2 - \cfrac{(5a)^2}{9 + (5a)^2}}}$	$a/(1 + a\uparrow 3/(2 - (5 \times a)\uparrow 2/(9 + (5 \times a)\uparrow 2)))$

Je patrna snaha po linearizaci zápisu, která vzniká proto, že vstupní zařízení počítače čte posloupnost znaků a nerozeznává jejich snížení nebo zvýšení. Znak $_{10}$ může být nahrazen libovolným jiným znakem nepoužívaným k jinému účelu.

V aritmetických výrazech se velmi často vyskytují některé elementární funkce, jako např. odmocnina, logaritmus, trigonometrické funkce apod. V Algolu jsou jisté identifikátory vyhrazeny pro označení několika *standardních elementárních funkcí*. Argument se píše za identifikátor vždy do kulaté závorky. Tak např. *abs* (E) značí absolutní hodnotu hodnoty aritmetického výrazu E, *sqrt* (E) druhou odmocninu. Význam označení *sin* (E), *ln* (E), *exp* (E), *sign* (E) je jasný. Narazí-li kompilátor při převádění programu do kódu počítače na některý z těchto vybraných identifikátorů, zařadí na příslušné místo takovou posloupnost instrukcí, která pak při výpočtu zapojí do funkce příslušný standardní podprogram, který počítá hodnotu použité funkce. Výraz $A + \sqrt{1 - \sin^2 B}$ se zapíše tedy ve tvaru A + *sqrt* $(1 -$ *sin* $(B)\uparrow 2)$,

výraz

$$\frac{e^y}{\ln (x^2 + 12.35\, y)}$$

ve tvaru $exp\,(y)/ln\,(x\uparrow 2 + 12.35 \times y)$.

Symbol :=, užitý v přiřazovacím příkazu, můžeme nazvat *přiřazovacím operátorem*. Nemá význam rovnosti v matematickém smyslu. Znaménka = pro rovnost se v Algolu též užívá, ale k jinému účelu. Činnost vyvolaná přiřazovacím příkazem se dá popsat asi takto: Výraz na pravé straně přiřazovacího příkazu se vyhodnotí s použitím běžných hodnot proměnných v něm vystupujících. Potom se běžná hodnota proměnné na levé straně nahradí právě vypočtenou hodnotou. Tedy činnost, která je následkem přiřazovacího příkazu v levém sloupci, se dá popsat těmito slovy (pravý sloupec):

$x := -.01$	Běžnou hodnotu proměnné x nahraď hodnotou $-1/100$.
$beta := gama + _{10} -3 \times delta$	Běžnou hodnotu proměnné *beta* nahraď běžnou hodnotou proměnné *gama*, zvětšenou o jednu tisícinu běžné hodnoty proměnné *delta*.
$n := n + 2$	Běžnou hodnotu proměnné n zvětšit o dvě.

Z posledního příkladu je patrné, že přiřazovací operátor nevyjadřuje rovnost v matematickém smyslu.

Kromě aritmetických operátorů užívá Algol ještě některých dalších operátorů, např. *relačních*. Relační operátory jsou znaménka používaná v matematice pro vyznačení rovnosti a nerovnosti čísel a výrazů. Užijeme-li ovšem např. relačního operátoru $<$ nebo $=$ pro vytvoření konfigurace $a < b$ nebo *aritmetický výraz* = *aritmetický výraz*, dostáváme tzv. *relaci*. Takto vzniklé relaci přiřazujeme pak *logickou hodnotu* **true** nebo **false** podle toho, zda relace platí nebo neplatí při běžných hodnotách proměnných, které se na jejích obou stranách vyskytují. Tedy např. relace $n + 1 \neq 99$ má hodnotu **false**, je-li běžná hodnota proměnné n rovna 98 a hodnotu **true** v ostatních případech.

Kromě dosud zmíněných *základních symbolů*, z nichž jsou sestavovány programy v Algolu, tj. kromě deseti cifer 0, 1, ..., 9, malých a velkých písmen latinské abecedy, operátorů, závorek a *oddělovačů* (např. $_{10}$, ale také ; : ,), používá Algol 60 mimo již uvedené logické hodnoty **true** a **false** ještě některých dalších anglických slov. Tato slova se při tisku sázejí polotučně a v rukopisu nebo strojopisu se podtrhávají. Svým významem, který mají v angličtině, upomínají na úlohu, kterou mají v Algolu. Jsou to např. **real, integer, begin, end, if, then, go to** a další. Každé toto slovo (nebo dvojice slov) má v Algolu význam jediného symbolu. Tento způsob zápisu symbolů byl zvolen ze dvou důvodů. Jednak nemá psací stroj zpravidla tolik různých znaků, které by mohly být k tomuto účelu do Algolu zavedeny, jednak se volbou slov

s vhodným původním významem zvyšuje názornost a přehlednost sestavených programů. Jak a kde se používá některých těchto symbolů a také oddělovačů, jako je středník, dvojtečka, čárka apod., ukážeme na příkladech.

Z ostatních druhů příkazů si na tomto místě uvedeme ještě *skokový příkaz* a *podmíněný příkaz*. Aby bylo možné jednotlivé příkazy v posloupnosti příkazů od sebe bezpečně oddělit, musí být konec každého příkazu nějak vyznačen. Nejčastěji ukončujeme příkaz středníkem. Uvidíme, že příkaz může být ukončen také symbolem **end** nebo **else.** Někdy potřebujeme během výpočtu zařadit vyplnění příkazů, které jsou umístěny na jiném místě programu. Musíme mít tedy možnost odkázat na příkaz, jímž tato část programu začíná. K tomu slouží *návěští*, které se klade před příkaz jím označovaný a odděluje se od něho dvojtečkou. Návěštím může být identifikátor nebo *celé číslo bez znaménka*. Příslušná sestava má tedy obvykle tvar

návěští : *příkaz*;

Přechod (předání řízení) na jiné místo programu se uskutečňuje skokovým příkazem, který má tvar

go to *návěští*;

Dojde-li se při plnění programu ke skokovému příkazu, plní se po něm příkaz označený příslušným návěštím a nacházející se někde v programu. Po něm se plní v přirozené posloupnosti příkazy za ním následující, dokud není tato posloupnost opět nějakým způsobem porušena.

Podmíněný příkaz má tvar

(2) **if** *relace* **then** *první příkaz* **else** *druhý příkaz*;

Má-li relace za **if** hodnotu **true,** plní se první příkaz a druhý příkaz se přeskakuje. Má-li relace za **if** hodnotu **false,** přeskakuje se první příkaz a plní se druhý příkaz. Potom se v obou případech plní příkazy následující za podmíněným příkazem, není-li ovšem příkaz, který se plní po prověření relace, skokovým příkazem předávajícím řízení na jiné místo programu. Druhý příkaz ve (2) může být opět podmíněným příkazem tvaru (2).

Podmíněný příkaz

(3) **if** $t = n - 1$ **then** $D := x \uparrow 2$ **else go to** special;

má stejný účinek jako podmíněný příkaz

(4) **if** $t \neq n - 1$ **then go to** *special* **else** $D := x \uparrow 2$;

a způsobí nahrazení běžné hodnoty proměnné D hodnotou výrazu x^2 (určenou při použití běžné hodnoty proměnné x) v případě, že běžná hodnota proměnné t je rovna hodnotě výrazu $n - 1$ (určené s použitím běžné hodnoty proměnné n). Potom se plní příkaz bezprostředně následující za naším podmíněným příkazem. Není-li běžná hodnota proměnné t rovna hodnotě aritmetického výrazu $n - 1$, předá se vyplněním skokového příkazu řízení té části programu, která začíná příkazem s návěštím *special*.

Vidíme, že v první variantě (3) našeho podmíněného příkazu je přiřazovací příkaz $D := x \uparrow 2$ ukončen znakem **else** a nikoliv středníkem, kdežto ve druhé jeho variantě (4) je tímto znakem ukončen skokový příkaz **go to** *special*.

V mnoha případech je zapotřebí, aby se namísto prvního nebo druhého příkazu v podmíněném příkazu (2) plnila celá posloupnost příkazů. Z těchto i z jiných důvodů se v Algolu považuje za příkaz i posloupnost jednotlivých příkazů (oddělených od sebe středníky a někdy též opatřených návěštími). Tato posloupnost však musí být uzavřena do tzv. *příkazových závorek* **begin** a **end**; nazývá se *složeným příkazem* a ten může být opatřen návěštím. Poslední příkaz složeného příkazu, který se nachází před **end**, nemusí být ukončen středníkem.

Význam použití složeného příkazu v podmíněném příkazu vysvitne z tohoto příkladu.

(5) **if** $x < b$ **then** $n := n + 1$ **else begin** $x := x + h$; **go to** *P*03 **end**;

Kdyby ve druhém (složeném) příkazu, který následuje za **else**, chyběly příkazové závorky, fungoval by podmíněný příkaz (5) správně pouze v případě, kdy relace za **if** má hodnotu **false**. Pak by se totiž po prověření relace plnil přiřazovací příkaz $x := x + h$ a po něm skokový příkaz **go to** *P*03, který by ovšem byl už za koncem podmíněného příkazu. V případě, že relace má hodnotu **true**, plnil by se po jejím prověření příkaz $n := n + 1$ a po něm opět skokový příkaz **go to** *P*03. Ten se však má v tomto případě při správném plnění podmíněného příkazu (5) přeskočit. Je totiž součástí složeného příkazu, který se plní vcelku, a to pouze v případě, že relace za **if** má hodnotu **false**.

Před sestavením nějakého jednoduššího programu zbývá nám promluvit ještě o jedné velmi závažné věci. Program v Algolu se sestavuje proto, aby byla podle něho řešena úloha na samočinném počítači. V jeho paměti musí být rezervována místa pro proměnné, které se budou v programu vyskytovat. Musí být tedy na začátku programu uveden seznam těchto proměnných, tj. jejich identifikátorů. Kromě toho jsou u některých počítačů ukládána celá čísla poněkud jinak (nebo na jiná místa paměti) než čísla necelá (racionální). Musí být tedy pro každou číselnou proměnnou vyznačeno, je-li to celé nebo necelé číslo.

Oba uvedené požadavky jsou splněny tzv. *popisem*, který se klade hned za **begin** ve složeném příkazu. Tím se stává ze složeného příkazu *blok*. Číselné proměnné jsou dvojího *typu*. Typ **integer** mají proměné, které jsou celými čísly v tom smyslu, jak jsme o tom mluvili výše. Ostatní číselné proměnné jsou typu **real**. Popis se začíná uvedením typu (**real** nebo **integer**), za nímž se vypíší identifikátory těch proměnných, které jsou vyznačeného typu. Jednotlivé identifikátory oddělíme mezi sebou čárkami a za poslední napíšeme středník. Může tedy popis vypadat takto:

real *a*, *b*, *x*, *sum*; **integer** *n*, *k*;

Většina proměnných v programech technických a numerických výpočtů je zpravidla typu **real**, neboť i když se na jistém stadiu výpočtu může vyskytnout průběžný výsledek,

který je celým číslem v obvyklém smyslu, umístíme ho mezi ostatní necelá čísla. Typ **integer** přisuzujeme zpravidla nesporným případům, jako jsou sčítací indexy nebo indexy u různých proměnných, označených stejným identifikátorem.

Tedy krátce shrnuto: Program psaný v jazyku ALGOL 60 je zpravidla blok. To znamená, že začíná symbolem (příkazovou závorkou) **begin**, za nímž následuje popis. Po něm následují jednotlivé příkazy, které mohou ovšem být také složenými příkazy nebo i bloky (tak zvanými subbloky hlavního programového bloku). Za každým z těchto příkazů je středník. Výjimku činí poslední příkaz, za nímž následuje příkazová závorka **end**, patřící k počátečnímu **begin.**

Uvedeme si nyní dva příklady na použití vyložených principů. V prvním sestavíme program pro numerický výpočet integrálu

$$\int_a^b \frac{(\sin x)\sqrt{x}}{x + e^x}\,dx$$

lichoběžníkovou metodou (viz [6]). Označíme $h = (b - a)/n$, kde n je počet stejných dílků, na které rozdělíme interval $\langle a, b\rangle$. Příslušný součet nahrazující hodnotu integrálu označíme identifikátorem *trapez*, takže platí

$$trapez = \tfrac{1}{2}h\{f(a) + 2\,[f(a + h) + f(a + 2h) + \ldots + f(b - 2h) + f(b - h)] + f(b)\}\,,$$

značí-li f integrovanou funkci. Zavedeme identifikátor *sum* pro označení součtu vytvářeného postupně sčítáním hodnot $f(a + h)$, $f(a + 2h)$ atd. až konečně přičtením hodnoty $f(b - h)$. Na začátku výpočtu položíme $sum = 0$. Příslušnou hodnotu budeme přičítat tak dlouho, dokud bude proměnná x, jejíž hodnoty vznikají postupně opakovaným přičítáním hodnoty h k číslu a, menší než b. Správný chod popsaného procesu zajistíme vhodným využitím podmíněného příkazu. Konečnou hodnotu součtu *sum* vynásobíme dvěma, přičteme k ní hodnoty $f(a)$ a $f(b)$ a výsledek vynásobíme nakonec číslem $h/2$. Je zřejmé, že toto všechno provede tento program (předpokládá se $a < b$):

```
begin real a, b, h, x, sum, trapez; integer n;
h := (b − a)/n; sum := 0; x := a + h ;
zde: sum := sum + sqrt(x) × sin(x)/(x + exp(x)) ;
x := x + h ;
if x < b then go to zde else trapez := h/2 × (sqrt(a) × sin(a)/(a + exp(a)) +
2 × sum + sqrt(b) × sin(b)/(b + exp(b)))
end
```

Přiřazovací příkaz, který funguje při postupném vytváření hodnoty proměnné *sum*, je opatřen návěštím *zde*, abychom na něj mohli předat řízení při příslušné hodnotě relace v podmíněném příkazu. Rozdělení programu na jednotlivé řádky

a právě tak případné vynechání volného místa nejsou podstatné. Užívá se jich pro zvýšení názornosti zápisu programu pro čtenáře. Operátory se na novém řádku neopakují.

Při podrobném prohlédnutí sestaveného programu nám padne do oka jedna neobratnost, ke které jsme přinuceni svými dosavadními znalostmi o programování v Algolu. Výpočet hodnoty integrované funkce je popsán třikrát na třech různých místech programu. Jazyk Algol poskytuje možnost vyhnout se tomuto opakování, a to zavedením tzv. *procedur*. Nemůžeme se jimi v tomto přehledném článku zabývat podrobně. Ostatně jsou v Algolu mnohé další věci, jako např. *indexované proměnné*, *boolovské proměnné* a *boolovské výrazy*, *přepínače*, *vlastní hodnoty* atd., o nichž zde nebudeme mluvit. Pro úplnější poučení se musí čtenář obrátit na příslušnou literaturu (hlavně [5], [6]). Zde si řekněme jen, že procedura v Algolu poskytuje možnost zformulovat pouze jednou nějaký výpočet, který se pak v programu provádí vícekrát s různými hodnotami *parametrů*. Příslušná posloupnost příkazů je opatřena *hlavičkou procedury*, která začíná jejím pojmenováním, tj. *identifikátorem procedury*. Tento *popis procedury* se umísťuje mezi popisy na začátku programu. Chceme-li na nějakém místě programu provést výpočet podle obecného schématu procedury, ovšem nyní s konkrétními, tzv. *skutečnými parametry*, stačí na toto místo dát *příkaz procedury*, tj. její identifikátor, za nímž následuje seznam skutečných parametrů uzavřený do kulatých závorek.

Algol připouští popis procedury v kódu počítače. To se hlavně týká úkonů, často ve většině programů opakovaných, jako je tisk průběžného nebo konečného výsledku nebo zastavení počítače. Použijeme-li pro takové procedury identifikátorů se zřejmým významem, stačí na příslušné místo programu napsat *print* (a) nebo *stop*. Po příkazu *print* $(a, b, \ldots, k)$ se buď vytisknou na papír nebo vyděrují do děrné pásky běžné hodnoty proměnných $a, \ldots, k$, tj. obsahy buněk, které jsou pro tyto proměnné rezervovány. Je-li v závorce aritmetický výraz, tiskne se hodnota tohoto výrazu pro běžné hodnoty proměnných v něm vystupujících. V našich příkladech nepoužijeme procedury zastavení. Považujeme výpočet za skončený, dospěje-li na konec zápisu programu.

Jako druhý příklad si sestavíme program řešení kvadratické rovnice $ax^2 + bx + c = 0$ s tiskem výsledků. Vezmeme v úvahu nejobecnější případ. Připouštíme i možnost $a = b = c = 0$. V tom případě nenecháme vytisknout nic. Může se totiž stát, že koeficienty a, b, c předem neznáme, že jsou např. počítány v nějakém širším programu, do něhož pak můžeme blok našeho programu zařadit jako subblok. Abychom vyznačili tiskem další možné výsledky, předpokládejme, že na buňce vyhrazené pro proměnnou s identifikátorem *inf* máme umístěnu kombinaci znaků, kterou čteme jako ∞. Na buňce vyhrazené proměnné s identifikátorem *im* máme kombinaci, kterou čteme jako i. Použijeme jí v případě komplexních kořenů, kdy tiskneme reálnou část, po ní i a pak imaginární část.

V programu použijeme „prohloubeného" podmíněného příkazu, jehož druhý příkaz (za **else**) je opět podmíněný příkaz. Povězme si napřed několik slov o funkci

takového příkazu. Má tvar

(6) **if** *relace* 1 **then** *P*1 **else if** *relace* 2 **then** *P*2 **else** *P*3 ;

Má-li relace 1 hodnotu **true**, plní se příkaz *P*1 a celý další zbytek podmíněného příkazu se přeskakuje. Má-li však relace 1 hodnotu **false**, přeskakuje se příkaz *P*1 a plní se příkaz, následující za prvním **else**. Ten je však také podmíněný a záleží tedy při jeho plnění na hodnotě relace 2. Je-li její hodnota **true**, plní se příkaz *P*2, kdežto *P*3 se přeskakuje. Je-li tato hodnota **false**, přeskakuje se *P*2 a plní se příkaz *P*3.

Příkaz *P*3 v (6) by mohl být opět podmíněný atd. Obecné schéma postupu pro takovéto složitější případy vzniká pokračováním právě provedené úvahy a dá se popsat takto: prověřují se postupně jednotlivé relace zleva doprava tak dlouho, až se přijde na první, která má hodnotu **true**. Pak se vyplní příkaz, který následuje za nejbližším **then** vpravo. Nemá-li ani jedna z relací hodnotu **true**, plní se příkaz, následující za posledním **else**.

Nyní uvedeme program.

```
begin real a, b, c;
if a ≠ 0 then go to A else
  if b ≠ 0 then begin print (−c/b, inf); go to END end else
    if c ≠ 0 then begin print (inf, inf); go to END end else
      go to END;
A: begin real d;
  d := b × b − 4 × a × c;
  if d < 0 then go to komplex else
    begin d := sqrt (d);
    if d = 0 then print (−b/(2 × a), −b/(2 × a)) else
      print ((−b + d)/(2 × a), (−b − d)/(2 × a));
    go to END
    end;
komplex: d := sqrt (−d); print (−b/(2 × a), im, d/(2 × a))
  end; END:
end
```

Složený příkaz s návěštím *A* je blok, neboť obsahuje popis proměnné *d*. Nepopsali jsme tuto proměnnou ve vnějším (programovém) bloku z toho důvodu, že se vyskytuje jen v jisté části výpočtu, která se popřípadě vůbec nemusí uplatnit. Měli bychom tedy jinak v programovém bloku zbytečně popsánu proměnnou, tj. měli bychom obsazenu buňku paměti, které třeba vůbec při výpočtu nepoužijeme. Popis ostatních proměnných *a*, *b*, *c*, provedený ve vnějším bloku, platí v našem případě i pro subblok s návěštím *A*. Proměnné popsané v subbloku, ztrácejí ovšem význam při výstupu z něho. Výhody těchto pravidel nejsou patrny u takového malého progra-

mu, jako je náš. Avšak tam, kde je třeba šetřit paměťovými místy, se výhody této tzv. blokové struktury projeví zřetelněji.

Mluvili jsme už o důvodech, proč musíme vytvořit složený příkaz (tj. uzavřít ho do příkazových závorek **begin** a **end**) z posloupnosti příkazů, která se má plnit, jestliže relace v podmíněném příkazu má hodnotu **false** (viz str. 163). Totéž jsme v našem programu učinili i v případě, že taková posloupnost příkazů následuje za **then**, tj. má se plnit, má-li relace hodnotu **true**. To je nutné kvůli pravidlům o plnění jiného druhu podmíněného příkazu. V něm je totiž uveden pouze příkaz, který se plní v případě, kdy relace má hodnotu **true**. V opačném případě se plní příkaz, který následuje bezprostředně za podmíněným příkazem. Mohla by tedy v tomto případě nastat kolize. Prostě řečeno, podmíněný příkaz by nebyl bez závorek **begin** a **end** vytvořen v souhlase s pravidly Algolu.

Na konec tohoto odstavce si ukážeme na dvou druzích *příkazu cyklu*, jak je možné programovat v Algolu cyklické výpočty. Tento příkaz může mít tvar

for *proměnná* := *počáteční hodnota* **step** *přírůstek* **until** *koncová hodnota* **do** *příkaz*;

Tato sestava říká, že příkaz následující za **do** se má plnit opakovaně pro všechny hodnoty té proměnné, jejíž identifikátor je za **for**, počínaje danou počáteční hodnotou, k níž pak postupně přidáváme přírůstek tak dlouho, dokud se nedostaneme ven z intervalu s krajními body *počáteční hodnota* a *koncová hodnota*. Této proměnné se říká *parametr cyklu*. Přírůstek může být záporný, tj. koncová hodnota může být menší než počáteční hodnota. Tak např. v důsledku příkazu

for x:= 20 **step** -2 **until** 10 **do** *print* $(x \uparrow 3)$;

se postupně tisknou třetí mocniny všech sudých čísel mezi 10 a 20 včetně (v pořadí podle klesajících hodnot).

Příkaz cyklu může mít také tvar

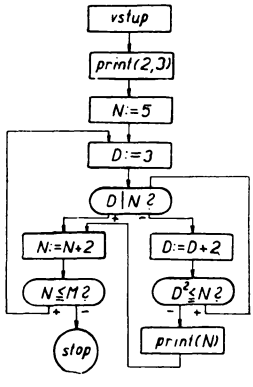
for *proměnná* := *arimetický výraz* **while** *relace* **do** *příkaz* ;

V relaci a v aritmetickém výrazu se vyskytují proměnné, jejichž hodnota se obecně při plnění příkazu mění. Může to být např. parametr cyklu samotný. Před vstupem do tohoto příkazu cyklu musí mít ovšem všechny proměnné vyskytující se v aritmetickém výrazu a v relaci přiřazenu hodnotu. Příkaz následující za **do** se plní pro tu hodnotu parametru cyklu, kterou dostaneme vyhodnocením aritmetického výrazu. Plní se ovšem opakovaně tak dlouho, pokud si relace zachovává hodnotu **true**. Jakmile nabude hodnoty **false**, vystupuje se z příkazu cyklu a plní se příkaz za ním následující. Může se stát, že se příkaz za **do** v této formě příkazu cyklu nevyplní ani jednou, má-li totiž relace už poprvé hodnotu **false**.

Do všech příkazů cyklu se dá vstoupit pouze přes jeho hlavičku začínající s **for**. To znamená, že není možné odkázat z vnějšku na nějaký příkaz, který je součástí příkazu cyklu. V tomto případě by totiž nebylo určeno, pro jakou hodnotu parametru cyklu se má příkaz provést. Příkaz následující za **do** může být opět příkaz cyklu nebo

podmíněný příkaz. Může to být také složený příkaz nebo blok, který obsahuje příkazy cyklu nebo podmíněné příkazy. Ukážeme si na následujícím příkladě, jak taková sestava funguje.

Máme za úkol naprogramovat vytisknutí všech prvočísel menších nebo nejvýše rovných kladnému číslu M. Zkoušku toho, je-li nějaké liché číslo $N > 0$ prvočíslem, můžeme provádět postupnými zkouškami dělitelnosti tohoto čísla lichými čísly D od 3 počínaje. Stačí se přitom omezit na čísla D nejvýše rovná odmocnině z N.



Sestavme si nejdříve blokové schéma programu. Pro dělitelnost čísla N číslem D užijeme obvyklého znaku $D \mid N$. Hodnotu **true** vyznačujeme znakem $+$, hodnotu **false** znakem $-$.

V blokovém schématu se vyskytují tři podmínkové (relační) bloky. Plnění příkazů, které závisí na relacích $N \leqq M$ a $D^2 \leqq N$, zajistíme vhodnými příkazy cyklu, které budou ovšem vloženy do sebe. Ke zjištění hodnoty relace $D \mid N$ použijeme s výhodou dělení pomocí operátoru $\div$. Obě proměnné, které se této operace účastní, musí být typu **integer**. Výsledek je opět typu **integer**. To je zajištěno tím, že namísto skutečné hodnoty podílu se vezme celé číslo, které vznikne z výsledku odseknutím podle vzorce

$$a \div b = sign\,\frac{a}{b}\left[\left|\frac{a}{b}\right|\right].$$

Zde $[r]$ značí celou část reálného čísla r, tj. největší celé číslo menší nebo nejvýše rovné r. Tedy např.

$$7 \div 4 = 1,\ 99 \div 100 = 0,\ 7 \div (-4) = -1\,.$$

Potom zřejmě relace $N - (N \div D) \times D = 0$ má hodnotu **true** tehdy a jen tehdy, platí-li $D \mid N$. V opačném případě má hodnotu **false**.

Program psaný v Algolu má tvar:

```
begin integer N, D, M ;
print (2, 3); D := 1 ;
for N := 5 step 2 until M do
  begin for D := D + 2 while D ↑ 2 ≦ N do
    if N − (N ÷ D) × D = 0 then go to END ;
  print (N); END: D := 1
  end
end
```

K funkci tohoto programu si musíme říci několik poznámek. Užili jsme v něm neúplného podmíněného příkazu, tzv. *příkazu „když"*, který má tvar

if *relace* **then** *příkaz* ;

Zmínili jsme se o něm již jednou v předcházejícím textu. Zde tedy připomínáme, že příkaz za **then** se plní v tom případě, jestliže relace má hodnotu **true**. V opačném případě se plní příkaz, který následuje za příkazem „když". V našem programu je však velice důležitá ta okolnost, že se tento příkaz „když" vyskytuje jako „řízený příkaz" ve vnitřním příkazu cyklu (který má parametr D). To má tento následek: Budiž N pevné; má-li relace $D \uparrow 2 \leqq N$ ještě hodnotu **true**, ale relace $N - (N \div D) \times D = 0$ má hodnotu **false** (tj. platí sice ještě $D^2 \leqq N$, ale D nedělí N), neplní se sice skokový příkaz **go to** *END*, ale neplní se v následujícím kroku ani příkaz *print* (N). Vnitřní příkaz cyklu totiž dosud funguje, a proto se zvětší D o dvě a prověřuje se znova relace $D \uparrow 2 \leqq N$. K výstupu z vnitřního příkazu cyklu dochází za dvou okolností. Buďto je už $D^2 > N$, a potom se podle pravidel o plnění příkazu cyklu neplní příkaz „když" následující za **do**; přejde se hned k příkazu procedury tisku čísla N; nebo je ještě $D^2 \leqq N$, ale D dělí N. Potom se plní skokový příkaz **go to** *END*, přeskakuje se příkaz procedury tisku a hodnota D se vrací na 1. V obou případech se dostáváme na konec složeného příkazu, řízeného vnějším příkazem cyklu (s parametrem N). To znamená, že N se zvětší o 2, a není-li dosud $N > M$, vstupujeme znova do vnitřního cyklu. Přesněji řečeno, vstupujeme do složeného příkazu, který je řízen vnějším cyklem a obsahuje vnitřní příkaz cyklu. Čtenář si může vyzkoušet funkci programu třeba na některých konkrétních číslech N. Podotkněme ještě, že Algol poskytuje také možnost vyhnout se zde užitému opakování přiřazovacího příkazu $D := 1$. K tomu bychom museli zavést ještě další pojmy.

3. Způsob, jakým bylo v předcházejícím odstavci podáno několik informací o jazyku ALGOL 60, se hodí pro úvod do programování v tomto jazyku. Je ho použito k výkladu např. v pracích [6], [7], [8]. K přesnému a vyčerpávajícímu popisu jazyka je v původní Zprávě (druhá část [5]) použito matematického upřesnění *syntaxe* živého jazyka. Syntax je tam popsána pomocí formálních tzv. *metalingvistických formulí*, nazývaných také *syntaktickými definicemi*.

Formální popis jazyka ALGOL 60 se opírá o tzv. *metalingvistické proměnné*, které odpovídají pojmu gramatické kategorie živého jazyka. Např. pojem *věta* můžeme při studiu každého živého jazyka vykládat dvojím způsobem. Buďto jím rozumíme gramatickou kategorii, která označuje uzavřenou skupinu slov vytvořenou s použitím příslušných gramatických pravidel a představující v jistém smyslu jednotku textu. Nebo jím rozumíme slovo, které samo může být součástí nějaké věty. „Hodnotou" gramatické kategorie „věta" je pak libovolná správně utvořená věta uvažovaného jazyka.

Mezi metalingvistické proměnné jsou zařazeny pojmy jako *základní symbol*, *aritmetický operátor*, *přiřazovací příkaz* apod. V syntaktických definicích jsou vyznačeny svým pojmenováním, ale uzavřeným do lomených závorek ⟨ ⟩, tedy např. ⟨identifikátor⟩, ⟨celé číslo⟩, ⟨příkaz cyklu⟩, ⟨podmíněný příkaz⟩ apod. Každá metalingvistická proměnná se vyskytuje na levé straně jedné syntaktické definice. Kromě metalingvistických proměnných a základních symbolů se v syntaktických

definicích používá *metasymbolů* : := a | . Druhý z nich má význam *nebo* ve formálně logickém smyslu.

Jednoduchým příkladem je syntaktická definice

$$\langle \text{relační operátor} \rangle : := < \mid \leqq \mid = \mid \geqq \mid > \mid \neq$$

která říká, že hodnotou metalingvistické proměnné ⟨relační operátor⟩ je každý ze šesti symbolů uvedených na pravé straně definice a oddělených navzájem znakem | .

Většina syntaktických definic však obsahuje na pravé straně opět metalingvistické proměnné. Při hledání některé z „hodnot" metalingvistické proměnné stojící na levé straně takové definice, tj. při sestavování příslušné posloupnosti základních symbolů (viz [5], druhá část, 1.1), dosazujeme za metalingvistické proměnné na pravé straně vhodné pravé strany příslušných syntaktických definic. Tento postup opakujeme tak dlouho, dokud nezískáme požadovanou posloupnost základních symbolů. Ukážeme si tento postup na příkladě.

Poněkud složitější syntaktickou definicí je definice

(7) ⟨identifikátor⟩ : := ⟨písmeno⟩ | ⟨identifikátor⟩ ⟨písmeno⟩ | ⟨identifikátor⟩ ⟨číslice⟩

Metalingvistická proměnná ⟨identifikátor⟩ vystupuje na obou stranách definice, a jde tedy o rekurentní definici. Říká především, že každé písmeno (malé nebo velké latinské abecedy) je identifikátor. Z druhé a třetí části pravé strany definice (7) vyplývá, že připojením písmene nebo číslice za toto písmeno dostaneme zase identifikátor. Takto vytvořené hodnoty metalingvistické proměnné ⟨identifikátor⟩ můžeme dosazovat opět do druhé a třetí části pravé strany (7) atd. Dostaneme tak postupně všechny možné posloupnosti vytvořené z písmen a číslic, které začínají písmenem. Takto zavedený pojem je tedy totožný s identifikátorem zavedeným ve druhé části tohoto článku.

Situace však může být mnohem složitější. Např. na pravé straně syntaktické definice, jejíž levá strana je ⟨aritmetický výraz⟩, se vyskytuje proměnná ⟨jednoduchý aritmetický výraz⟩. Pro ni platí syntaktická definice

⟨jednoduchý aritmetický výraz⟩ : := ⟨člen⟩ | ⟨operátor typu sčítání⟩ ⟨člen⟩ | ⟨jednoduchý aritmetický výraz⟩ ⟨operátor typu sčítání⟩ ⟨člen⟩

Dále je

⟨člen⟩ : := ⟨činitel⟩ | ⟨člen⟩ ⟨operátor typu násobení⟩ ⟨činitel⟩

⟨činitel⟩ : := ⟨prvotní výraz⟩ | ⟨činitel⟩ ↑ ⟨prvotní výraz⟩

Na pravé straně syntaktické definice, jejíž levá strana je ⟨prvotní výraz⟩, se vyskytuje opět proměnná ⟨aritmetický výraz⟩.

Podrobným rozborem syntaktických definic se zjistí, že existují takové třídy metalingvistických proměnných, že od každého prvku třídy je možno po určitém počtu použití vhodných syntaktických definic dospět právě naznačeným způsobem k libovolnému prvku téže třídy ([10]). Jedna z těchto tříd obsahuje 26 proměnných, mezi

nimiž je právě ⟨aritmetický výraz⟩, ⟨činitel⟩, ⟨prvotní výraz⟩, ale také ⟨skutečný parametr⟩, ⟨relace⟩, ⟨zápis funkce⟩ atd. V práci [9] najde čtenář důkaz toho, že přitom nejde o definici bludným kruhem, ale o simultánní definici jednotlivých proměnných ve smyslu definice obecně rekurzívních funkcí. Autorka ukazuje, že tyto funkce jsou zde primitivně rekurzívní. Čistě matematický popis množin hodnot jednotlivých metalingvistických proměnných podává práce [10]. O principu matematického přístupu k syntaxi jazyka viz např. [11].

Formální syntax je ve Zprávě doplněna řadou pravidel, která berou v úvahu též obsahovou stránku „frází" jazyka ALGOL 60. Tato pravidla je nutné zachovávat při sestavování programů, procedur atd. V důsledku toho není dosud zcela vyjasněna otázka, pod jaký typ jazyků popsaných v [11] patří ALGOL 60 a jak ho formalizovat v celé jeho úplnosti, např. pomocí syntaktických definic nebo jiným „matematickým" způsobem.

Literatura

[1] Jiří Hořejš: Principy samočinných číslicových počítačů. PMFA *7*, 15 (1962).

[2] A. J. Kitov: *Elektronické číslicové počítače*. SNTL, Praha 1960.

[3] А. А. Ляпунов: П логических схемах програм. Проблемы кибернетики *1*, 46 (1958). Физматгиз, Москва.

[4] *Автоматизация программирования*. Физматгиз, Москва 1961.

[5] J. W. Backus: *Programování v jazyku* ALGOL 60. SNTL, Praha 1963.

[6] Daniel D. Mc Cracken: *A Guide to Algol Programming*. John Wiley, New York-London 1962.

[7] B. Higman: What EVERYBODY should know about ALGOL. The Computer Journal *6*, 50 (1963).

[8] Г. Боттенбрух: Структура АЛГОЛ-60 и его использование. ИЛ, Москва1963.

[9] Rózsa Péter: Primitiv — rekursive Wortbeziehungen in der Programmierungssprache ALGOL 60. A Magyar tudományos akadémia, Matematikai kutató intézetének közleményei, VI. évfolyam, A. sorozat, 1—2. füzet, 1961.

[10] Karel Čulík: Formal Structure of ALGOL and Simplification of its Description. Symbolic Languages in Data Processing, Gordon and Breach, New York-London 1962.

[11] N. Chomsky: On Certain Formal Properties of Grammars. Inf. and Control *2*, 137 (1959). (Ruský překlad v Кибернетический сборник *5*, 279 (1962), ИЛ, Москва.)

Elektrizace indických železnic

probíhá rychlým tempem. Anglická firma BICC dostala od r. 1958 devět zakázek na elektrizaci více než 2250 km trati střídavým proudem o napětí 25 kV.

Ivan Soudek

Největší lineární urychlovač elektronů na světě

se staví v novém centru atomového výzkumu u Charkova. Urychlovací trubice je 240 m dlouhá a dává elektronům energii až 2000 MeV.

Ivan Soudek