ARexxManuál

pêeloùil Stiftero

COLLABORATORS			
	TITLE : ARexxManuál		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	pêeloùil Stiftero	July 10, 2022	

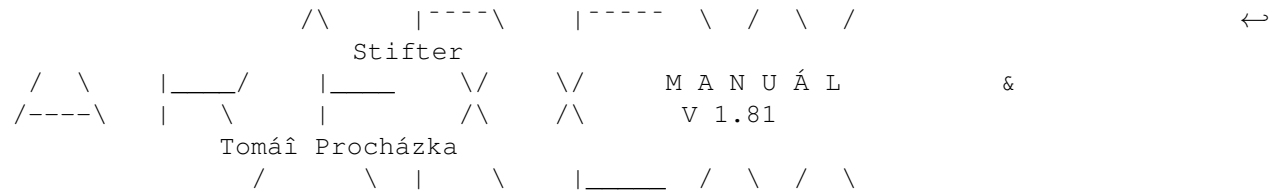
REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

ARexxManuál

1.1 ARexx - Pêíruçka pro uùivatele A R E X X U Verze 1.81 (12.2.2000)



1.2 Historie vývoje manuálu

-> H I S T O R I E <-

Verze 1.8 (18.8.1999)
- více informací níùe

Verze 1.81 (12.2.2000)
- manuál pêeloùen do kódování ATO-E2 (v rámci podpory tohoto nového standartu çeského kódování pro Amigu)
- opravena drobná chyba v pomocném programu "Programy.rexx"
- také jsem opravil spoustu pravopisných chyb,kterých je na kaùdém kousku spousta. Chtëlo by to celý manuál znovu pêeçíst a zamëêit se na chyby. Pokud o manuál projevíte zájem, bude jeho vývoj urçitë pokraçovat...

Podrobnëjîí popis událostí související se vznikem verze 1.8

Çervenec 1998

Uplynulo pár let od doby, co svët spatêil pûvodní verzi tohoto manuálu tento manuál dostal do ruky jeden poçítaçový nadîenec, který stále vëêí v lepîí budoucnost Amigy. A kdyù loni v létë poznal ARexx, stal se jedním z jeho nejvëtîích pout k Amize. Jelikoù se anglicky zaçal uçit aù pozdëji byl odkázán na to, co vyçetl z çasopisû a pêeluîtil z angliçtiny. Proto vêele pêivítal návod v çeîtinë. Vzal v úvahu autorovu ùádost a pokusil se manuál trochu zpêíjemnit, s nadëjí, ùe se najde více takových nadîencû, protoùe nadîenci jsou právë ti, díky kterým se Amiga udrùela pêi ùivotë i v tëch nejtëùîích çasech a snad to budou také ti, kteêí se zúçastní jejího znovuzrození.

...

A tak se také stalo.

Záêí 1999 (Verze 1.8)

*** Tohle je první "kompletní" vydání upravené verze manuálu ***

Jsem rád, ùe se vám upravený manuál dostal do rukou a doufám, ùe vám bude prospëîný tak jako më. Jen më mrzí, ùe je tento manuál trochu sloùitý a mnohdy tëùko srozumitelný. Originál, z nëhoù je tento manuál pêeloùen, asi nebyl urçen pro naprostého zaçáteçníka v oboru programování. Pokusil jsem se výklad místy zjednoduîit, doplnit poznámkami. Nëkteré çásti jsem dokonce napsal znovu a hlavnë jsme toho spoustu pêidal. Ale nebylo v mých silách a ani jsem neusiloval napsat celý manuál znovu.

Pokud jste jiù nëkdy programovali nebo alespoñ máte trpëlivost a tuîení co to programování je , urçitë pro vás nebude ARexx ùádnou pêekáùkou. Já si myslím, ùe ARexx patêí do kategorie VELMI snadných programovacích (skriptových) jazykû. Kaùdopádnë vám doporuçuji zaçít od první kapitoly, protoùe se zpoçátku (kapitoly 1. - 3.) dozvíte informace, které jsou dûleùité pro pochopení dalîích çástí. Doplnil jsem manuál o spoustu odkazû, ale ty první têi kapitoly si stejnë radîi pêeçtëte, protoùe se na në aù tak pêíliî neodkazuji. Odkazy jsou zamëêeny pêedevîím k provázání dalîích çástí...

Ti, kteêí jiù mají starou verzi manuálu, jistë potëîím, protoùe jsem opravil nëkolik nepêesností, chyb a doplnil chybëjící funkce... (a urçitë i nové chyby... jak se znám...)

Bohuùel nejsem specialista pêes pravopis, proto se omlouvám za pêípadné pravopisné chyby. Doufám, ùe jiné neù pravopisné se nenajdou a pokud ano, tak doufám ùe bude v mých silách je odstranit.

Budoucnost

Já osobnë nepovaùuji tuto verzi ARexx Manuálu za finální, jeîtë se toho jistë dá mnoho zlepîit a navíc jsme nemël ças stihnout udëlat vîe, co jsem zamýîlel. Dalîí zlepîení budou ale pêeváùnë záviset na vás, uùivatelích, zda se vám bude manuál líbit a budete chtít, abych pokraçoval v jeho vývoji. Nemëlo by smysl, kdybych se dalîí mësíc zabýval nëçím, co nikdo nechce. Já ale vëêím, ùe budete z mou dosavadní prací alespoñ trochu spokojení a pokud ne, seùeñte si pûvodní verzi manuálu, a mûùete ji zkusit upravit sami....

Snad se ARexx, ale i samotná Amiga bude dále rozvíjet a dokáùe svëtu, ùe stojí za pozornost a obdiv ... jednou se tak jistë stane ... (a uù brzy)

Tomáî Procházka

1.3 Informace o më...

-> A D R E S A <-

Tomáî Procházka
Masarykova tê. 955
Orlová-Lutynë
735 14

E-mail:
tom.p@atlas.cz
WWW: http://freeweb.bohemia.net/atom nebo http://freeweb.coco.cz/atom

Uvítám jakékoliv nápady a podnëty k zdokonalení...

Pokud më budete chtít informovat o chybë, klidnë se na më obraïte a to Stále platí i o spoustë pravopisných chyb.

Dëkuji.

A nezapomeñte: AMIGA FOREVER THE BEST !!!

1.4 Upozornëní ohlednë mé E-mailové adresy.

Internet se vyvíjí a mëní den ze dne, proto v pêípadë zmëny mé e-mailové dresy navîtivte mé stránky, kde se dozvíte novou. Poçítám, ùe se tak brzy stane...

Pokud by se náhodou zmënila také adresa mých WWW stránek (coù je nepravdëpodobné), mûùete vyuùít sluùeb pro vyhledávání osob jako jsou:

lide.seznam.cz nebo ldap.atlas.cz

1.5 Podëkování...

Díky: Rodiçûm za pochopení, Commmodore za Amigu a ARexx, gymnáziu za nepochopení, Bohu za Jeho lásku a podporu, Omariimu za likvidaci kníùky, u4ia za skvëlý moduly a vîem ostatním, kteêí se nedoùili vydání kníùky.....

TP: A já dëkuji, za to, ùe se naîel nëkdo, kdo pêeloùil tento manuál a pomohl tak mnohým "neangliçtináêûm" pêi výuce toho jazyka. A pomohl také më, proto je tato nová verze odmënou...

1.6 Copyright k manuálu

Váùení a milí, tëîí më, ùe tohle vûbec çtete, ale na úvod vás budu muset trochu zprudit. Pêednë: Zpenëùeníchtivé jedince nakopu do êiti, jelikoî toto dílo je pro kaùdýho z a d a r m i k o ù. Právo na komerçní zneuùítí mám za prvý ich a za druhý to musí odsouhlasit vlastník práv na originál (Commmodore, Escom...). Ofîem pokud si tenhle dokument necháte pro svoji potêebu, máte za úkol, napsat mi nëjakej ten Email, vyçíslit chyby, podëkovat, nebo jen tak pokec. This is not public domain !!! This is pub domain...

stifter@mira.cz

Podotýkám, ùe tenhle spis bol pêeloùen nëkdy roku 94 a çeîtina odpovídá mému techdejîùímu psychostavu. Chyb jako trávy, neobratné konstrukce atd. Keby se naîel voljaký obratný floutek, který bi si chtël dát ten work a pêedëlat tento manuál do formátu AmigaGuide popê. jiné hypertextové ûchilnosti, contactujte më na výîe zmíněné Emailovce nebo na SnailMail:

Stiftero
Husitská 634
284 01 Kutná Hora

Zadarmo máte ARexxový manuál. Nevím co byste chtëli výc. Víc Vám taky nedám.

Dobêe se bavte !

1.7 Co je ARexx ? This is the question...

Vítejte v ARexxu

Pokud çtete tyto êádky, jistë nepatêíte mezi pouhé hráçe (paêany). Jestli-ùe tomu opravdu tak je, urçitë jste se jiù se slovem ARexx setkaly. ARexx totiù podporuje kaùdý lepîí program a to tak, ùe má arexxový port.
Co je to arexxový port ? Bylo by pêedçasné na tuto otázku odpovëdët,

ale mohu vám êíci, ùe pokud ARexxu porozumíte, objevíte novou, dosud skrytou sílu vaîeho poçítaçe ...

A teð, k vlastnímu popisu ARexxu. ARexx je amigácká verze PCçkové programovacího jazyka "Rexx", jenù byl pûvodnë vyvinut v laboratoêích IBM a byla urçen pro îirokou vrstvu uùivatelû, u kterých se nevyùadují hlubîí programátorské znalosti... V PC svëtë se vîak tento programovací jazyk pêíliî neuplatnil, zatím co na Amize se stal velkým standardem...

A co z toho plyne pro nás ? Máte k dispozici výborný programovací jazyk, jenù vyniká znaçnou jednoduchostí a zároveñ uspokojujícím, nëkdy dokonce neoçekávaným výsledkem. Avîak musím vás srazit opët k zemi. Naçekejte, ùe si sednete k poçítaç i a za pár hodin zplodíte nového DOOMA. Na takové vëci totiù ARexx není stavëný.
Co tedy dovede ? Umoùñuje vám individuální pêestavbu vaîeho pracovního prostêedí. Hodí se obzvláîtë dobêe na skripty, které êídí a modifikují aplikace a definují jejich interakci (spolupráci) s ostatními aplikacemi. Z toho vyplývá, ùe pokud vám v textovém editoru chybí nëkterá funkce, váî grafický program neumí operaci, jenù çasto potêebujete nebo jen chcete pêeformátovat text, tak právë vám je tento programovací jazyk urçen.
A co k tomu potêebuji ? Po pravdë êeçeno. Témëê nic. ùádné velké investice. Vîe, co potêebujete se nachází ve vaîem systému. Tedy pokud vlastníte Workbench 2.0 (to je ale dnes snad samozêejmé) a výîe (Funguje i na niùîích verzích.) Manuál máte pêed sebou. A nyní potêebujete uù jen pêedvídavost, chuï do experimentování, trochu pêirozené inteligence... a mûùete
zaçít
.

A co do budoucna? Brzy vám jistë pêestane staçit,to co naleznete v tomto manuálu. Zklamu vás, ale pokud neumíte anglicky,uù se asi nic víc nedovíte, ale pokud patêíte mezi ty îïastnëjîí, mûùete vesele pokraçovat dál, protoùe, ARexx je prakticky nekoneçný. S kaùdou novou knihovnou pro ARexx nebo s novým programem je ARexx vetîí a mocnëjîí...(viz. Ukázkový obrázek)

Kde mám hledat nëco víc ? Podívejte se na
referenci
!

1.8 Poznávání ARexxu...

Výuka A R E X X U

Tento manuál vám nabízí uvedení do ARexxu,popisuje vytváêení arexxových programû (maker) a obsahuje referençní informace k arexxovým programûm...

Kapitola 1.
Úvod do ARexxu
Tato kapitola nabízí pêehled ARexxu, zpûsob ↩
fungování
na Amize a nejdûleùitëjîí funkce programovacího jazyka.

Kapitola 2.
První kroky

Zde je popsáno, kam se ukládají arexxové programy a jak se volají. Navíc jsou zde nëkteré programové pêíklady.

Kapitola 3.
Prvky ARexxu
Tato kapitola popisuje detailnë pravidla a koncepty, na kterých se ARexx zakládá.

Kapitola 4.
Pêíkazy
Tato kapitola obsahuje abecední seznam arexxových pêíkazû a jejich popis. Pod pêíkazy se rozumí pokyny k akci.

Kapitola 5.
Funkce
Tato kapitola popisuje pouùití funkcí (ARexxem pouùívané programové pokyny , které vracejí výsledek.), obsahuje abecední seznam integrovaných arexxových funkcí a externích funkcí knihovny RexxSupport.

Kapitola 6.
Pomoc pêi testování
Tato kapitola popisuje funkce pro hledání chyb na úrovni zdrojového textu (source-level debugging), které mohou být pouùity hlavnë pêi vývoji a testování programû.

Kapitola 7.
Syntaxní analýza
Tato kapitola popisuje získání informaçních vzorû z êetëzcû.

Pêíloha A.
Chybová hláîení
Tato pêíloha obsahuje seznam arexxových chybových hláîení.

Pêíloha B.
Kommandové pomocné programy
Tato pêíloha obsahuje seznam arexxových kommand, která mohou být volána ze Shellu. (Vzhledem k faktu, ùe v nëmçinë i angliçtinë je pro "povel" pouùit výraz "kommando", proto jsem jej pouùil také - pozn.)

1.9 Typografické konvence

Dokumentové konvence
==========

V manuálu platí následující konvence:

KLÍÇOVÁ SLOVA	Klíçová slova jsou napsána velkými písmeny; u argumentû jsou ovîem pouùita i malá písmena.
<výraz>,<znak>	Potêebné argumenty jsou psány malými písmeny.
<n>	Promënné, výrazy a vîe ostatní co není souçástí pêíkazu nebo funkce, ale co doplñujete vy. Místo îpiçatých závorkách napíîete jakoukoliv symbol (promënnou), êetëzec...
|	Alternativní moùnosti výbëru jsou od sebe oddëleny svislou çarou.
{ }	Potêebné alternativy stojí v sloùených závorkách.
[ ]	Volitelné souçásti pêíkazu stojí v hranatých závorkách.
Klávesa1-Klávesa2	Klávesová kombinace s pomlçkou (-) znaçí souçasný stisk kláves.
Klávesa1,Klávesa2	Klávesová kombinace s çárkou (,) znaçí postupný stisk kláves (za sebou).
Klávesy Amiga	Na klávesnici napravo a nalevo od mezerníku se nachází klávesy na provádëní speciálních funkcí
->	tato îipeçka je pouùitá v pêíkladech, znamená výstup programu nebo jen jednoho pêíkazu, çi funkce, který je ve skuteçnosti vypsán na obrazovku nebo jinak dále zpracován

---

Dalîí rady:

Vùdy se êiðte pravidlem "Praxe je nejlepîí uçitel", pokud si danou vëc nevyzkouîíte, nemusíte ji plnë pochopit....

1.10 Seznam doplñkové literatury, knihoven a programû

Zdroje dalîích informací a doplñkû k ARexxu

Následující knihy obsahují dalîí informace k naučení a pouùívání ARexxu:

Modern Programming Using REXX
R.P.O'Hara & D.G.Gomberg
Prentice-Hall, 1985.

The REXX Language: A Practical Approach to Programming
M.F.Cowlishaw
Prentice-Hall, 1985.
(vyîlo i v nëmçinë)

Programming in REXX
Charles Danea
J.Ranade IBM Series

Using ARexx on the Amiga
Chris Zamara & Nick Sullivan
Abacus, 1991.

Amiga ROM Kernel Reference Manual Libraries, Third Edition
Addison-Wesley, 1992.

Pêístupnëjîí je ale snad Aminet.

Staçí si jen nechat vyhledat "REXX" a budete pêekvapeni, kolik toho najdete. Urçitë si kaùdý z vás vybere. I ti, co kromë Amigy mají i jiné koníçky, napêíklad Radioamatéêi... (InfrARexx,...)

Zvláîï vám doporuçuji:

ARexxGuide od Robina Evanse nebo ARexxGuide od South West Amiga Group

MUIRexx - umoùní vytváêet aplikace v MUI prostêedí. (Vyùaduje MUI)
- obsahuje také editor pro snadný návrh !
TritoRexx - také slouùí k vytváêení grafického prostêedí...
GUI4CLI - nesouvisí pêímo z ARexxem, ale GUI zde lze také vytváêet
- obsahuje také editor pro vytváêení grafického prostêedí

Knihovny:
RexxReqTools - umoùní pouùít základní requestry (file,string,...)
RexxDosSupport - obsahuje 12 DOSovských pêíkazû
RexxArpLib - dovede velice zajímavé vëci, napê. KRESLIT
RexxTricks - obsahuje spoustu funkcí rûzných zamëêení, jejím pouùitím získáte funkce AmigaDOSu, SCSI funkce, Funkce pro práci s clipboardem, funkce pro práci s polem (têídění,vyhledávání, vkládání), funkce pro práci z veêejnými obrazovkami, ikonami a také funkce pro zjiîïování typû souborû a kódování souborû.

Názorný pêíklad vám mûùe poskytnout následující obrázek, na kterém uvidíte nëkolik pêíkladû grafických prostêedí (MuiRexx,TritonRexx,Gui4Cli,RexxArpLib)

Ukázkový obrázek - ukonçí se pêes "Esc"

1.11 1. kapitola

Úvod do ARexxu

Programovací jazyk ARexx slouùí jako centrální uzlový bod, pêes který si aplikace - i od rûzných výrobcû - mohou vymëòovat data a povely. S ARexxem je napêíklad moùné sestavit telekomunikaçní paket tak, aby zvolil elektronickou schránku (mailbox), poùadovaná finançnë-technická data vytáhl, uloùil, a pak je bez zásahu uùivatele automaticky nechal vyhodnotit v tabulkovém procesoru.

ARexx je interpretující jazyk, který pouùívá ASCII-Data jako vstup. arexxový interpret je program RexxMast. Nachází se ve Workbenchi

v adresáêi "System". RexxMast hlídá provádëní arexxového programu. Kdyù pêi pêekladu nebo provádëní êádku narazí na chybu, pêeruîí provádëní programu a zobrazí
chybové hláîení
. Toto interaktivní testování je
na jednu stranu pomoc pêi uçení, na druhou stranu ulehçuje tvorbu popêípadë opravu programû, protoùe chybová hláîka bezprostêednë sdëlí, na kterém místë doîlo k chybë.

Dalîí informace o ARexxu:

1.1
Základní znalosti
1.2
ARexx na Amize
1.3
Zvláîtní vlastnosti ARexxu

1.12 Základní pêedpoklady pro práci s ARexxem

1.1 Na jakou cílovou skupinu se obrací ARexx ?

arexxové programy a skripty nevyùadují ùádné detailní znalosti Amigy, nicménë by jste mëli mít základní znalosti následujících bodû:

* Otevêení Shellu a zadávání pêíkazû AmigaDOSu

* Práce s textovým editorem (ED, MEmacs, lépe vîak CED,GoldED,...)

* Vytvoêení uùivatelského startupového souboru (user-startup)

Neù mûùete jít na mënění nebo vytváêení arexxových skript, mëli by jste znát základy pracovního prostêedí Amigáckého Workbenche a AmigaDOSu. Zkuîení uùivatelé Amigy zjistí, ùe se s ARexxem pracuje efektivnëji a lehçeji neù s AmigaDOSem. ARexx mûùe být také nasazen, na doplnění nebo nahrazení AmigaDOSových pêíkazû nebo AmigaDOSových skript, jako i na vytváêení integrovaných aplikací.

1.13 ARexx na Amize

1.2 ARexx na Amize

ARexx je podporován na vîech hardwarových konfiguracích. Od Workbenche 2.0 je ARexx integrovanou souçástí Amigy. ARexx pouùívá dvë velmi dûleùité funkce Amigy - MULTITASKING a KOMUNIKACI PROCESû.

Multitasking je schopnost provádët souçasnë nëkolik programû. Mûùete napêíklad souçasnë formátovat disk (DF0:), hrát hry, editovat soubor aj.

Pod komunikací procesû (Interprocess Communication - IPC) se rozumí schopnost poçítaçe, vyměňovat si informace mezi právë bëùícími aplikacemi.

Komunikace procesû se odvíjí pêes takzvané Message-porty (nëco jako rozhraní, funguje na principu poîtovní schránky). Message-port je adresa obsaùená v aplikaci, pêes kterou mohou být zprávy a kommanda posílány nebo pêijímány. Kaùdý Message-port má jméno, a pêenos zprávy na aplikaci vyùaduje zadání jména Message-portu do arexxového skriptu.

Operace pêi vysílání a pêíjmu zpráv se odvíjejí v následujícím poêadí:

1. Pêi inicializaci otevêe aplikaçní program Message-port.

2. Aplikace çeká na pêíjem zprávy.

3. Amigácký systém sdëlí aplikaci, ùe zpráva dorazila.

4. Aplikace vykoná akci, obsaùenou ve zprávë.

5. Aplikace sdëlí odesílateli (ARexxu), ùe zpráva byla pêíjmuta a zpracována.

Tento zpûsob pêenosu zpráv není ohraniçen jen na aplikaci a ARexx. Více aplikací mûùe si posílat, pêijímat, vymëòovat zprávy pêes ARexx jako pêes ústêednu. Pêirozenë musí být zúçastnëné aplikace kompatibilní s ARexxem.

1.14 Zvláîtní vlastnosti ARexxu

1.3 Zvláîtní vlastnosti ARexxu

Programovací jazyk ARexx se vyznaçuje následujícími vlastnostmi

* ùádné datové typy - S daty je nakládáno jako s individuelními znakovými êetëzci. Nedeklarují se hodnoty proměnných. To znamená, ùe nemusíte pêedem definovat kaùdou proměnnou a urçovat o jaký typ se bude jednat. S çíselnými proměnnými je moùno manipulovat jako s êetëzci. ARexx prostë není ùádny Pascal...

* Interpretované provádëní - Interpret ARexxu, program pêímo çte a v reálném çase provádí, programy se tudíù nekompilují. Je to sice pomalejîí, ale pêesto pouùitelné. Moùná ùe se nëkdy doçkáme i kompilátoru...

* Automatická správa prostêedkû - Díky automatickému internímu reservování paměti jsou uù nepotêebná data a znakové êetëzce odstranëny.

* Prûbëhové a událostní monitorování a hledání chyb - Prûbëhové popêípadë událostní monitorování umoùñuje speciální oîetêení chyb, kvûli kterých by se jinak program pêeruîil. Pomocné testovací funkce umoùñují pêezkouîení celého programu, coù znaçnë redukuje çasovou nároçnost na vývoj a testování.

* Knihovny funkcí - Externí knihovny funkcí obsahují rozîíêené, pêedprogramované funkce, které vám umoùní neustále rozîiêovat mnoùství dostupných funkcí...

1.15 2. kapitola

```
                První kroky

V této kapitole jsou popsány následující akce:

 *
                 Startování interpretu ARexxu
                   A
                 Automatický start ARexxu
                   A
                 Manuální spouîtëní ARexxu
                 *
                 Informace k arexxovým programûm
                   A
                 Volání arexxových programû
                    Na  závër této kapitoly se podíváme na pár pêíkladû nejedná se ↩
                        o výuku,
ale  pêedevîím  o  to,  aby  jste  se  mohli  udëlat pêedstavu o arexxových
programech.   Bliùîí  popis  vîech  çástí  ARexxu  se  dozvíte  v  dalîích
kapitolách.
    Klidnë  mûùete  následující çást pêeskoçit a vrátit se k ní aù pozdëji,
já osobnë bych ji zde vûbec nedával, ale byla zde v pûvodní verzi, tak jsem
ji  zde  nechal.  Na  druhé  stranë jsem ji doplnil odkazy, takùe pokud se
nechcete zabývat teorií, mûùete se hned pustit do programování. Upozorñuji,
ùe  odkazy ale nejsou natolik kompletní, aby jste se nauçili úplnë vîechno,
odkazy jsou jen na to, çím se daný pêíklad zrovna zabývá.

 *
                 Pêíklady programû
```

1.16 Startování interpetu ARexxu

```
                2.1  Startování ARexxu (program RexxMast)


   Pro  práci  s  ARexxem je potêeba aktivovat arexxový interpret.  Jedná se
o program  RexxMast,  který  se  nachází  v  adresáêi  "System"  ve  vaîem
Workbenchi.

  To je moùné  buð
                 automaticky
                 nebo
                 manuálnë
                .  Pêi kaùdém startu nebo
zastavení ARexxu je vypsána hláîka.
```

1.17 Automatický start ARexxu

2.1.1 Automatický start ARexxu

ARexx mûùete spustit dvëma zpûsoby. Kdyù ikonu RexxMast pêesunete do adresáêe WBStartup nebo kdyù zmëníte user-startup sequenci.

Pêidejte do svého user startupu (s:user-startup) êádek run >nil:REXXMAST

Pêi dalîím bootování (novém startu systému) jiù bude ARexxový interpret spuîtën automaticky a vy se nemusíte o nic starat.

1.18 Manuální spouîtëní ARexxu

2.1.2 Manuální spouîtëní ARexxu

Mûùete buð dvakrát kliknout na ikonu RexxMast nebo spustit RexxMast ze Shellu. Uùivatelé disketových jednotek a starîích systémû mohou uîetêit pamëï, kdyù spouîtëjí ARexx jen v nutném pêípadë. (Doufám, ùe uù se dnes nenajdou takoví, kterým by chybëla ta trocha pamëti.)

Ze Shellu (CLI) spusïte pêíkazem "REXXMAST" lépe "run RexxMast"

1.19 Informace k arexxovým programûm

2.2 Informace k arexxovým programûm

arexxové programy jsou zapisovány normálnë do adresáêe REXX:. Je vîak moùné je zapisovat zcela kdekoliv, ale pouùití adresáêe REXX: má má své výhody:

* Program mûùe být volán bez zadání cesty
* Vîechny arexxové programy jsou pohromadë
* Vëtîina aplikací hledá arexxové programy v adresáêi REXX:

Pokud dodrùíte urcité konvence, ulehcíte si správu programû. Napêíklad je dobré zapisovat arexxové programy s pêíponou .rexx. Dají se lépe rozliîit od ostatních programû.

Poznámka: Stalo se mi, ùe program bez koncovky nefungoval správnë !

1.20 Volání arexxových programû

2.2.1 Volání arexxových programû

Program
RX
slouùí k volání (spuîtëní) arexxového programu. Kdyù je zadána cesta, hledá ARexx na zadané cestë. Kdyù nezadáte cestu, potom hledá v aktuální adresáêi a adresáêi REXX:.

Dokud je program zapsán v adresáêi REXX: , není potêeba, aby mël Pêíponu .rexx. Pêíkazu "RX Program.rexx" pak odpovídá "RX Program".

Krátký program mûùe být zadán i v pêíkazové êádce, kdyù je zadán do uvozovek. Následující program poîle pët souborû na tiskárnu, ale musí být zadán na jedné êádce.

```
RX "DO i=1 TO 5 ; ADDRESS command 'copy myfile.' || i 'prt:'; END"
```

Kdyù je nëjaká aplikace kompatibilní s ARexxem, mohou být z této aplikace volány arexxové programy. K tomu zvolte poloùku z menu nebo zadejte urçité kommandové podmínky. Bliùîí informace çerpejte z dokumentace k aplikaci. Arexxové programy mûùete spouîtët i z Workbenche. Pro tento úçel vytvoête ikonu (radëji spouîtëjte programy ze Shellu, uîetêíte si práci). Do ikony zadejte: Sys:Rexxc/RX (staçí jen RX) - RX spustí automaticky RexxMast, pokud jiù nebëùí.

Dále mûùete do ikony zadat následující tooltypy:

* Console=<výstupní zaêízení> - pêesmërování výstupu napê.: Con:0/0/640/256/Example/Close
* CMD=<název programu> - bude spuîtën program s jiným názvem neù má ikona.

1.21 Pêíklady programû

2.3 Pêíklady programû

Následující pêíklady ukazují, jak mûùete pouùít ARexx k zobrazení textových êetëzcû na monitoru, provedení výpoçtû a k aktivování rutiny pro kontrolu chyb.

Programy mohou být zadány pêes jakýkoliv textový editor (Ed, Memacs aj.) nebo program na zpracování textu. Pêi pouùití programu na zpracování textu zapiîte váî program ve formátu ASCII. ARexx podporuje rozîíêenou sadu ASCII (ß, ö, $\times$, Þ, A, aj.). Tyto znaky jsou rozeznány jako obyçejné znaky a jako takové korektnë konvertovány z malých písmen na velká. To vîak nëplatí pro çeské kódování a korektní pêevod vîech çeských znakû!

Pêíklady ukazují pouùití nëkterých syntaktických poùadavkû napê.:

* Komentáêové êádky
* Pravidla na odstup znakû
* Rozliîování na malá a velká písmena

* Pouùití jednoho nebo dvou uvozovacích znamení (uvozovky)

Kaùdý arexxový program sestává nejménë z jedné êádky (popisující program) a z jednoho pêíkazu. Arexxové programy musí vùdy zaçínat

komentáêem

. Úvodní znaky /* (vùdy musí být ukonçen zanky */) sdëlí RexxMast interpretu, ùe se jedná o arexxový program. Bez znakû /* */ se ARexx k programu nehlásí (není arexxový program, neznám, nechci, jdi pryç... pozn.aut.). Po spuîtëní programu ignoruje ARexx vîechny dalîí komentáêe v souboru. Komentáêe jsou dûleùité pro srozumitelnost programu, protoùe pêibliùují logiku a strukturu programu.

2.3.1 Amiga.rexx

Tento program ukazuje, jak je slovo SAY pouùito v arexxovém pêíkazu k zobrazení textu. Pêíkazy jsou pokyny, které blíùe urçují provádënou akci. Kaùdý pokyn zaçíná

symbolem

, v naîem pêípadë slovem SAY. (Symboly jsou zmëněny na velká písmena v prûbëhu programu). Po SAY následuje

êetezec

. Pod êetëzcem se rozumí êada znakû uzavêených mezi apostrofem "'" nebo uvozovkami """.

Program 1. Amiga.rexx Spusï program !

```
! Poznámka !
/*Jednoduchý program*/

SAY
'AMiGA , multimediální poçítaç nejvyîîí kvality.'
```

Zadejte výîe uvedený text a uloùte jej pod jménem REXX:Amiga.rexx . K spuîtëní programu otevêete Shell a zadáte

```
RX
Amiga
```

Cesta a jméno jsou sice REXX:Amiga.rexx, ale k spuîtëní nepotêebujete cestu ani pêíponu .rexx, pokud je program zapsán skuteçnë v adresáêi REXX:

Ve vaîem Shellovém oknë by se mël objevit text:

```
Amiga , multimediální poçítaç nejvyîîí kvality.
```

2.3.2 Stáêí.rexx

Tento program vyzve k zadání dat a poté je naçte.

Program 2. Stáêí.rexx Spusï program !

```
! Poznámka !
/*Spoçítat stáêí ve dnech*/
```

```
                SAY
                'Prosím, prosím, moc prosím, zadejte Váî vëk:'

                PULL
                stari
SAY 'Vy jste asi ' stari*365 'dnû starý.'
```

Nyní zapiîte program pod názvem REXX:Stáêí.rexx a spustïe jej pêíkazem RX Stáêí

Tento program zaçíná komentáêem (1.êádek), který popisuje úkol programu. Vîechny arexxové programy zaçínají komentáêem. Pêíkaz SAY slouùí k zobrazení výzvy k zadání dat. V pozdëjîích kapitolách se dozvíte, ùe existuje mnohem elegantnëjîí zpûsob jak zobrazit výzvu k zadání dat pomoci pêíkazu

```
                OPTIONS
                  Pêíkaz
                PULL
```

çte zadané hodnoty - v naîem pêíkladu stáêí. PULL pêeçte zadané hodnoty, pêemëní je na velká písmena a uloùí do promënné. Promënné jsou symboly, kterým mûùe být pêiêazena hodnota. Doporuçuje se, volit trefná jména promënných. Zde je pro úschovu hodnoty zadané uùivatelem pouùito jméno "stari".

```
Poznámka: V názvech promënných lze pouùít jen tyto znaky: A-Z,a-z,$,_,!,@,#
          a pro pole promënných jeîtë znak '.' ( Polozka.cislo ). Promënná
          "cislo" pak udává jakoby çíslo êádku v tabulce o n êádcích.
          Viz. kapitola zabývající se "
                 symboly
                ".
```

Poslední êádek násobí promënnou "stáêí" hodnotou 365 (pêestupné roky jsou ignorovány) a pêíkazem SAY zobrazí výsledek. Vîimnëte si, ùe promënná "stáêí" se nemusí deklarovat, protoùe její hodnota je v okamùiku pouùití ve výrazu pêezkouîena (pêíklad dat bez datových typû). Kdyù chcete vidët, co se stane, kdyù nezadáte çíslice, spusïte program a zadejte písmena. Objeví se hláîka s çíslem êádku a popisem druhu chyby.

(viz.

```
                chyová hláîení
               )
```

Pokud jste jiù tak neuçinili, zkuste programu místo svého vëku zadat nëjaký text. Moùná, ùe budete pêekvapeni.

2.3.3 Calc.rexx

V tomto programu si pêedstavíme pêíkaz

```
                DO
               , s kterým mohou být programové
```

pokyny opakovány. Nakoukneme pod pokliçku exponenciálnímu operátoru "**" (exponent se vztahuje k mocninë). Napiîte program ve svém oblíbeném textovém editoru nebo DTP-programu a zapiîte ho pod jménem REXX:Calc.rexx . Pouùijte Kommando "

```
                RX
                Calc" k spuîtëní programu.
```

```
 Program 3.  Calc.rexx        Spusï program !
                ! Poznámka !
               /*Výpoçet druhých a têetích mocnin.*/

                DO
                i = 1 TO 10    /*Zaçátek smyçky - 10 cyklû*/

                SAY
                i i**2 i**3    /*Výpoçet a výstup*/

                END
                       /*Konec smyçky*/
SAY 'Hotovo.'
```

Pêíkaz DO zpûsobuje, ùe se pokyny mezi pêíkazy DO a END pravidelnë opakují. Promënná "i" je indexová promënná pro smyçku a zvyîuje se pêi kaùdém opakování (iteraci) o 1. çíslo, které následuje po symbolu "TO", je koncová hodnota pro pêíkaz DO a mohla by tam místo konstanty být promënná nebo celý výraz.

Vîeobecnë pouùívají arexxové programy mezery mezi rûznými çástmi pêíkazu. V programu 3 se ovîem nepouùívají mezery mezi exponenciálním znamením a promënnými a konstantami.

Vîimnëte si, ùe jsou pokyny ve smyçce posunuty. Programovací jazyk to nevyùaduje, ale pro srozumitelnost a pêehlednost programu je to dûleùité (konec a zaçátek smyçky je vidët na první pohled).

2.3.4 Sudé.rexx

Pêíkaz

```
                IF
```

umoùñuje provedení pokynu pod urçitou podmínkou. V pêedloùeném pêíkladu jsou çísla od 1 - 10 klasifikována jako sudá nebo lichá. Jsou dëlena dvojkou a zbytek po dëlení je pêezkoumán. Aritmetický

```
                operátor
                // spoçítá zbytek po dëlení.
```

```
 Program 4.  Sudé.rexx        Spusï program !
                ! Poznámka !
               /*Sudé nebo liché*/

                DO
                i = 1 to 10      /*Poçátek smyçky - 10 cyklû*/

                IF
                i // 2 = 0 THEN typ = 'sudé'

                ELSE
                typ = 'liché'

                SAY
                i 'je' typ

                END
```

```
                              /* Konec smyçky */
```

Êádek s podmínkou IF nám udává, ùe v pêípadë, kdyù pêi dëlení vyjde nula jako zbytek pêi dëlení promënné "i" dvojkou, oznaçí çíslo jako sudé a výsledek uloùí do promënné "typ". V pêípadë, ùe zbytek není nula, pêeskoçí program vëtev THEN a vykoná místo toho vëtev

```
                ELSE
               , coù znamená oznaçení
```

çísla za liché.

2.3.5 Mocnina.rexx

Tento pêíklad pêedstavuje koncept

```
                funkce
               , to znamená skupinu pokynû,
```

které mohou být provedeny ve vhodných souvislostech. S pomocí funkcí mûùete sestavit rozsáhlé a komplexní programy z malých modulû (stavebních kamenû programu). Funkce pêipouîtëjí také pouùití toho samého programového textu pro podobné operace v rûzných çástech programu.

Funkci pêedstavuje výraz (jméno), po kterém následuje otevêená závorka (mezi jménem a závorkou není mezera). Jeden nebo více výrazû, které se nazývají argumenty, mohou následovat. Po posledním argumentu následuje uzavírající závorka. Skrz argumenty jsou pêedány informace ke zpracování danou funkcí. (Pouùitá funkce se nazývá

```
                Interní
               , tj. ta, kterou si
```

progra- mátor sám definuje.)

Program 5. Mocnina.rexx Spusï program !

```
                ! Poznámka !
               /* Definice a volání funkce */

                DO
                i = 1 to 5

                SAY
                i Mocnina(i)     /*Volání funkce mocnina*/

                END

                EXIT
                           /*Ukonçení programu*/

Mocnina:       /*Jméno funkce*/

                ARG
                x        /*Vyvolat argument*/

                RETURN
                x**2     /*Umocnit a vrátit zpët*/
```

Smyçka s indexní promënnou "i" zaçíná na DO a konçí na END. Promënná i se pokaùdé zvýîí o 1. Smyçka má pët cyklû. Smyçka obsahuje výraz, který vyvolá funkci "Mocnina" pêi vyhodnocení výrazu. Výsledek funkce je vpysán pêíkazem SAY.

Funkce "Mocnina" je definovaná pêíkazy

```
ARG
a
RETURN
a poçítá druhou
```

mocninu. ARG vyvolá hodnotu z argumentu "i", RETURN pêevádí výsledek funkce zpët na pêíkaz SAY.

Okamùitë jak je funkce volána smyçkou, hledá program funkçní jméno "mocnina:", vyvolá argument "i", provede poçetní operaci, a vrátí se na zaçátek smyçky. Pêíkaz

```
EXIT
```

ukonçí program po posledním prûbëhu smyçky.

2.3.6 Výsledky.rexx

Pêíkaz

```
TRACE
slouùí k êizení
monitorování
.
```

Program 6. Výsledky.rexx Spusï program !

```
! Poznámka !
/*Pêedstavení monitorace výsledkû (sledování êádku po êádku)*/
TRACE results
sum = 0 ; sumq = 0 ;

DO
i = 1 to 5
   sum = sum + 1
   sumq = sumq + i**2

END
SAY 'sum=' sum 'sumq=' sumq
```

V oknë uvidíte kaùdý prûbëh smyçkou, provádëní êádek, vçetnë výsledkû. Bez pêíkazu

```
TRACE
by byl ukázán jen koncový výsledek.
```

2.3.7 Známka.rexx

Tento program poçítá koncovou známku pro urçitého ùáka na základë dosaùených známek ve çtyêech písemkách a jednom ústním zkouîení. Prûmërná známka z písemky ç.1 a ç.2 má pêispët 30% ke koncové známce, písemka ç.3 a ç.4 má pêispët 45%, a zkouîení 25%. Dávejte si pozor na psaní americké desetinné teçky místo çárky.

Poté co program ukáùe výslednou známku, zeptá se uùivatele, zda chce jeîtë spoçítat dalîí známku. Odpovëd je naçtena (PULL). Není-li odpovëð "q", je pokraçováno ve smyçce. Je-li odpovëd "q", je smyçka opuîtëna a program ukonçen.

```
 Program 7.  Známka.rexx        Spusï program !
                 ! Poznámka !
                /*Program na výpoçet známky*/
SAY "Nazdar, já ti moùná spoçítám známky (jen kdyù budeî moc hodnej)."
response = 0

                 DO
                 while response ~ = "q"
   /*Smyçka, dokud odpovëð není q*/
   SAY "Prosím zadejte vîechny známky"
   SAY "Písemka 1:"

                 PULL
                 es1
   SAY "Písemka 2:"
   PULL es2
   SAY "Písemka 3:"
   PULL es3
   SAY "Písemka 4:"
   PULL es4
   SAY "Zkouîení :"
   PULL p
   Znamka = (((es1 + es2)/2*.3) + ((es3 + es4)/2*.45) + (p*.25))
   SAY "Koncová Známka je " Znamka
   SAY "Chcete pokraçovat ?  (Ukonçit = Q)."
   PULL response

                 END

                 EXIT
```

1.22 3. kapitola

```
                Prvky programovacího jazyka ARexx
```

Tato kapitola jedná o pravidlech a konceptu programovacího jazyka ARexx. Zároveñ bude objasnëno, jak ARexx interpretuje znaky a slova, pouùívaná v programech. Zde popisované prvky:

```
 *
                 Token
                 - nejmenîí çást jazyka ARexx

 *
                 Klausule, dodatky
                 - nejmenîí proveditelná jednotka srovnatelná s vëtou

 *
                 Výrazy
                  - skupina vyhodnocených tokenû.

 *
                 Kommandové rozhraní
                 - (povelové rozhraní) - proces, pêes který se
```

arexxové programy domlouvají s aplikacemi (sluçitelnými s ARexxem).

Kapitola dále obsahuje popis provozního prostêedí ARexxu. Tento oddíl se zamëêuje na zkuîené uùivatele Amigy a obsahuje technické detaily k procesové komunikaci.

3.5 Provozní prostêedí
Poznámka: Nëkterými çástmi se budeme blíùe zabývat nëkdy v dalîích kapitolách. Zde popsáné vëci jsou vîak nezbytný základ pro pochopení dalîích çástí...

1.23 Tokeny

3.1 Token

Tokeny jsou nejmenîí oddëlené jednotky programovacího jazyka ARexx (slova). Mohou sestávat z jednoho nebo více znakû. Tokeny mûùeme rozdëlit do pëti kategorií:

*
Komentáêe
*
Symboly
(promënné)
*
Êetëzce
*
Operátory
*
Zvláîtní znaky

1.24 t-komentare

3.1.1 Komentáêe

Kaùdá skupina znakû, která zaçíná znaky /* a konçí */ , tvoêí komentáê. Kaùdý arexxový program musí zaçínat komentáêem. Ke kaùdému /* musí být i odpovídající */ . Pêíklad:

```
/*Z.T.S. Software sucks*/
```

Komentáêe mohou být umístëny kdekoliv v programu a jakkoliv zasunuty do sebe. Pêíklad:

```
/*Zasunutý /* komentáê*/ */
```

Doporuçuje se vybavit program poçetným komentáêem - umoùní Vám i jiným uùivatelûm lépe pochopit funkci a úçel programu. Interpreter ignoruje komentáê pêi çtení programových dat - komentáêe nezpûsobují ùádné zdrùení

programu.

1.25 t-symboly

3.1.2 Symboly (Promënné)

Pod symbolem se rozumí libovolná skupina znakû a-z A-Z 0-9 dále pak znaky . _ ! ? $ @ #. Kdyù interpret çte program, pêeloùí si symboly na velká písmena. Symbolu AMiGaRuLeZ odpovídá symbol AMIGARULEZ. Jsou dva druhy symbolû:

Pevné symboly
Skupina povolených znakû, která vîak zaçíná çíslem (0-9) nebo teçkou(.). Hodnota symbolu je vùdy symbol sám, pêeloùen na velká písmena. Nejçastëjîím pevným symbolem je çíslo, které vîak mûùe obsahovat znak "E" pro definici exponentu, napê. 3E8 je 300 000 000.

Promënné symboly
Skupina povolených znakû, která zaçíná abecedním znakem (A-Z,a-z) nebo nëkterým ze znakû ! $ _ @ #. Dále vîak mûùe obsahovat i çísla. Promënný symbol se také mûùe chovat jako pevný, dokud mu není pêiêazena hodnota.

Promënní symboly mohou dále tvoêit:

Kmenové symboly	Promënné symboly, které vîak konçí teçkou. Napê. "Kmen."
Sloùené symboly	Promënné symboly, které obsahují více teçek. teçek. Napê. "Kmen.vetev."

Promënné, sloùené a kmenové symboly jsou oznaçovány jako promënné. Bëhem programu jim mûùe být pêiêazena hodnota. Dokud není promënná obsazena hodnotou, není tzv. inicializována. Pro neinicializované promënné je jako hodnota pouùita promënná sama (ve velkých písmenech).

Poznámka:
Nëkterým arexxovým funkcím se pêedávají jako parametr znak nebo skupiny znakû,napêíklad u funkce

```
OPEN()
,je jako poslední argument nëktrý ze znakû
```

W, R, A. Pokud je zadáte takto, nic se nestane, protoùe neinicializovaný symbol má hodnotu stejnou jako svûj název. Nesmí se ale stát, ùe pêedem napê. symbolu W pêidëlíte nëjakou hodnotu, protoùe to by vedlo k chybë ! Pokud jako takový parametr zadáte jako

```
êetëzec
'W' nemûùe k chybë nikdy
```

dojít. Záleùí jen na vás zda si dokáùete ohlídat názvy pouùívaných promënných nebo budete pouùívat êetëzce.

Popis sloùených a kmenových symbolû
Sloùené a kmenové symboly mají zvláîtní vlastnosti, které jsou urçeny obzvláîtë pro výstavbu polí (promënných) a seznamû. Kmenové symboly umoùñují inicializaci celé têídy sloùených symbolû. Jeden sloùený symbol má strukturu Kmen.n1.n2.n3.n4.n5.n6.n7.n8.n9...nk ,kde výchozím jménem je kmenový symbol a kaùdý uzel n1...nk je pevný nebo jednoduchý symbol. Kdyù

je kmenovému symbolu pêiêazena hodnota, pêiêadí se (z jeho strany) tato hodnota vîem od kmenového souboru odvozeným, sloùeným symbolûm. Hodnota sloùeného symbolu závisí také na pêedchozích pêiêazeních na tento symbol nebo na pêísluîný kmen.

Kdyù se objeví sloùený symbol v programu, je jeho jméno rozîíêeno, tím, ùe je kaùdý uzel nahrazen jeho aktuální hodnotou. Poêadí znakû hodnoty mûùe být libovolné z libovolných znakû (vçetnë mezer) a nemëní se na velká písmena. Výsledek rozîíêení je nové jméno, pouùívané namísto sloùeného symbolu. Kdyù má napêíklad J hodnotu 3 a K hodnotu 7, je sloùený symbol A.J.K rozîíêen na A.3.7.

Sloùené symboly mohou být brány jako forma míst v pamëti (bufferû), závislých na pêiêazení nebo obsahovë adresovatelných (asociativních). Vezmëme si, ùe chcete uloùit êadu telefonních çísel a jmen a následovnë vyvolat. Normální postup by byl, uloùit jména a çísla do dvou polí. Çíslo by jste hledali tím zpûsobem, ùe by jste procházeli pole, dokud by jste nenarazili na zadané jméno napê. Jméno.12, jehoù telefonní çíslo by bylo pêeçteno z druhého pole.

Kdyù pracujete se sloùenými symboly, tak by symbol NAME mohl obsahovat volané jméno a NUMMER.NAME by bylo rozîíêeno na odpovídající telefonní çíslo napê. NUMMER.CBM

Sloùené symboly mohou být pouùity jako normální indexovaná pole, a nabízí pêitom výhodu, ùe je jen jeden pêíkaz (na kmen) potêeba, aby bylo celá skupina inicializována. Program v následujícím pêíkladu pouùívá kmeny "nummer." a "adr." k vytvoêení elektronického adresáêe.

Program 8. TelNr.rexx Spusï program !

```
                ! Poznámka !
               /*Telefonní adresáê k objasnëní sloùených promënných*/

                IF

                ARG()
                ~ = 1 THEN DO
   SAY "Pouùití - RX TelNr jméno"

                EXIT
                5
END
/*Pro zobrazení telefonních çísel/adres otevêít vlastní okno*/
CALL
                OPEN
               (out,"con:0/0/640/60/ARexx telefonní seznam")
IF ~ result THEN DO
   SAY "Otevêení okna se nepovedlo ... Sorry"
   EXIT 10
   END
/*Definice çísel*/
nummer. = '(není nalezeno)'
nummer.wsh = '(555) 001-0001'
adr. = 'nenalezena'
nummer.CBM = '(555) 002-0002'
adr.CBM = '1200 Wilson Dr., West Chester, PA 19380'
```

```
/*Vlastní práce uù je hotová*/

                ARG
                name   /*Jméno*/

                CALL

                WRITELN
                out,name || "Tel.ç. je " nummer.name
CALL WRITELN out,name || 'Adresa je ' adr.name
CALL WRITELN(out,"Stisknëte enter pro opuîtëní programu.")
CALL READLN out
EXIT
```

K spuîtëní programu otevêete Shellové okno a zadejte

```
                RX
                TelNr cbm
```

Objeví se okno s adresou a telefonním çíslem Commmodore (pravdëpodobnë se jedná o starou adresu).

Vîimëte si zadání funkce

```
                WRITELN()
```

. Pokaùdé je zadána trochu jinak,ale vîe je správné a plnë funkçní. Doporuçuji si vám vîak doporuçit pouùití posledního zápisu se závorkami, protoùe u nëj je nejjistëjîí, ùe interpet pozná, ùe se jedná o funkci a navíc nemusíte pouùít pêíkazu

```
                CALL
```

.

Mûùe se vám vîak stát, ùe pouùijete závorky a vynecháte pêíkazu CALL a i pêesto, ùe by mëlo být vîe v poêádku, nastane chyba. Informace o tomto nepêíjemném jevu nalezenete u popisu pêíkazu

```
                CALL
```

.

1.26 t-retezce

3.1.3 Êetëzce

Êetëzec je libovolná skupina znakû uzavêená mezi znaky "'" nebo """. Na konci êetëzce musí být vùdy stejný znak, jako byl pouùit na zaçátku. Tyto znaky vîak nesjou souçásti êetëzce, jen jej ohraniçují, aby arexxový interpet dokázal êetëzec rozpoznat.

Pokud má být nëkterý z uvozovacích znakû sám souçástí êetëzce, musí být zadán dvakrát ('' nebo "").

Pêíklad:

```
 SAY 'Zde je apostorf: '' to je paráda !'
```

lze to vîak provést jeîtë takto:

```
 SAY "Zde je apostrof: ’ to je paráda !"
```

Hodnota êetëzce je êetëzec sám. Délka êetëzce závisí na poçtu znakû v êetëzci. Kdyù v êetëzci nejsou znaky, oznaçuje se jako prázdný êetëzec. Délka jednoho êetëzce je omezena na 65 535 znakû, coù je myslím celkem uspokojivé mnoùství pro vëtîinu aplikací.

Hexadecimální a binární êetëzce

Kdyù bezprostêednë po êetëzci následuje X nebo B, jedná se o hexadecimální nebo binární êetëzec. Potom by se mël skládat z hexadecimálních, popêípadë binárních çíslic.

```
’4A 3B C0’X   /* Tohle jso têi znaky */
’00110111’B   /* 8 bity je dáno jedno písmeno */
```

Pro pêehlednost jsou v ARexxu dovoleny mezery na hranicích bytu (byte - 8bitû). Jednosuîe êeçeno, v hexadecimálním êetëzci mûùou být mezery kaùdé dva znaky, zatímco v binárním êetëzci kaùdých 8 znakû.

Hexadecimální a binární êetëzce se hodí pro znaky nedefinované v ASCII a pro strojové specifické informace, napê. adresy. Obratem jsou pêemënëny na komprimovanou strojovou vnitêní formu.

Poznámka: Pokud zadáte ùetëzce tak, jak bylo výîe uvedeno, interpret tyto êetëzce pêevede vùdy do znakové podoby (Podle ASCII tabuly.).

Pêíklad:

```
Text = ’41 68 6F 6A 20 6C C1 73 6B 6F’X   /* Promënná Text nyní obsahuje

                SAY
                Text                    textový êetëzec.            */
```

Program vypíîe -> Ahoj lásko

1.27 t-operatory

3.1.4 Operátory

Operátory jsou tvoêeny s tëchto znakû: ~ + - * / = < > & | ^.

Rozliîujeme çtyêi druhy operátorû:

*
Aritmetické operátory
- vyùadují jeden nebo dva numerické operandy a dávají numerický výsledek.

*
Zêetëzovací operátory
- spojují dva êetëzce do jednoho. (Konkatenace)

*
Porovnávací operátory
- vyùadují operandy a dávají booleovský výsledek, tj. "0" nebo "1".

*
Logické operátory
- vyùadují jeden nebo dva booleovské operandy a
dávají booleovský výsledek.

Kaùdému operátoru je pêiêazena priorita, která urçuje poêadí provádëní operací ve výrazu. Operátory s prioritou vyîîí (8) budou vykonány pêed operátory s niùîí prioritou (1).

3.1.4.1 Aritmetické operátory

Dûleùitá têída operátorû jsou çíselné operátory a çísla, skládající se ze znakû (0-9) (.) (+) (-) (mezera). Pêi vypsání çísla v exponenciálním tvaru následuje po çísle "e" nebo "E" a jedno celé çíslo (se znaménkem).

K zadání çísel mohou být pouùity
êetëzce
nebo
symboly
. Protoùe ARexx
nemá typy dat, nemusí být promënné pêedem deklarovány. Místo toho je kaùdý êetëzec pêezkouîen pêi zpracování, jestli náhodou není çíslo. Následující pêíklady pêedstavují platná çísla: 33 , 12.3 , 0.321e12 , +15.

Mezery pêed a po çísle jsou pêípustné. Mezery mohou stát mezi znaménkem a çíslem, ale ne v çísle samotném.

Základní pêesnost pouùívaná pêi aritmetických operacích mûùe být zmënëna pêi bëhu programu. Poçet platných míst, pouùitých pêi aritmetických operacích, se êídí podle nastavení Numeric Digits a mûùe být zmënën pêíkazem
NUMERIC
.

Poçet výsledných desetinných míst závisí na operaci a desetinných místech zúçastnëných operandû. ARexx pouùívá koncové nuly k pêedvedení pêesnosti výsledku. Kdyù je celkové çíslo vëtîí neù nastavení Numeric Digits, tak ARexx zformátuje çíslo do exponenciálním tvaru. K tomu má následující moùnosti:

* Vëdecký zpûsob výpisu - exponent je takový, ùe nalevo od desetinné çárky stojí jen jedno çíslo.

* Technický zpûsob výpisu - çíslo je pêepracováno tak, ùe exponent je násobkem têí a çíslo nalevo od desetinné çárky je 0-999.

Následující tabulka ukazuje seznam aritmetických operátorû:

Aritmetické operátory

Operátor	Priorita	Pêíklad	Výsledek
+ (pêedznamení)	8	+'3.12'	3.12
- (záporné pêedznamení)	8	-"3.12"	-3.12

** (mocnění)	7	0.5**3	0.125
* (násobení)	6	1.5*1.50	2.250
/ (dělení)	6	6/3	2
% (celoçíselné dělení)	6	-8%3	-2
// (zbytek po dělení)	6	5.1//0.2	0.1
+ (sçítání)	5	3.1+4.05	7.15
- (odçítání)	5	5.55-1	4.55

3.1.4.2 Zêetëzovací operátory - operátory spojující êetëzce (Konkatenace)

ARexx definuje dva zêetëzovací operátory. První (|| - dvë svislé çáry) spojuje dva êetëzce do jednoho, aniù by mezi në byla vloùena mezera. Tento druh zêetëzení mûùe být zadán implicitnë. Kdyù jsou
symbol
a êetëzec
zadány bez mezery, chová se ARexx, jako by tam operátor || byl. (Napê. stari'let') Druhá zêetëzovací operace je oznaçena pouze mezerou. Dá dohromady dva operandové êetëzce a vloùí mezi në mezeru.

Vîechny spojovací operace mají prioritu 4. Tabulka shrnuje rûzné operace.

Operátor	Operace	Pêíklad	Výsledek
||	Zêetëzení	'Rád bych,'||'ale...'	Rád bych,ale...
mezera	Zêetëzení s mezerou	'Dobrý' 'den.'	Dobrý den.
ùádný	Implicitní zêetëzení	jedna'dva'tri	JEDNAdvaTRI

Poznámka: Proměnné "jedna" a "tri" nejsou definované.

Upozornění: Poslední moùnost spojení êetëzcû (bez operátoru) nefunguje s proměnnými jejichù název je tvoêen pouze jedním
symbolem
(znakem). Napê: "Say A','B" vyhodí tuto chybu:
+++ Error 8 in line x: Unrecognized token
Ale "SAY AA','BB" je jiù v poêádku.

3.1.4.3 Porovnávací operátory

ARexx podporuje têi druhy porovnávání.

* exaktní - porovnává po znacích.
* êetëzcové - vedoucí mezery jsou ignorovány, do kratîího êetëzce jsou mezery doplněny.
* çíselné - operandy se na základë pêednastavení Numeric Digits pêemění do vnitêního çíselného formátu. Následovnë je provedeno aritmetické porovnání.

Porovnání dává vùdy booleovský výsledek a çísla 0,1 pêedstavují booleovské "False" a "True". Kdyù je oçekáván booleovský výsledek,pouùití

jiné hodnoty neù "0" a "1" vede k chybë. Jako booleovská hodnota je akceptován jakýkoliv ekvivalent napê. 0.000 nebo 0.1.E1 .

S vyjímkou operátorû pro exaktní rovnost (==) a exaktní nerovnost (~==) urçují vîechny porovnávací operátory dynamicky, zda budou porovnávány êetëzce nebo çísla a çíselné porovnávání nastane v pêípadë, ùe oba operandy jsou platná çísla. Jinak budou operandy porovnávány jako êetëzce.

Vîechna porovnávání mají prioritu 3. Následující tabulka ukazuje seznam povolených porovnávacích operátorû.

Porovnávací operátory

Operátor	Operace	Mód
==	Exaktní rovnost	Exaktní
~==	Exaktní nerovnost	Exaktní
=	Rovnost	êetëzcový/çíselný
~=	Nerovnost	êetëzcový/çíselný
>	Vëtîí neù	êetëzcový/çíselný
>= nebo ~<	Vëtîí nebo rovno	êetëzcový/çíselný
<	Menîí neù	êetëzcový/çíselný
<= nebo ~>	Menîí neù nebo rovno	êetëzcový/çíselný

3.1.4.4 Logické (booleovské) operátory

ARexx definuje çtyêi logické operace NE, A, NEBO a exklusivní NEBO. Vîechny tyto operace potêebují booleovské operandy a dávají booleovský výsledek. Pokus, pracovat s jinými neù s boolskými operandy, vede k chybë. Následující tabulka ukazuje seznam povolených logických operací.

Logické operátory

Operátor	Priorita	Operace
~	8	Logické NE (otoçení) - NOT
&	2	Logické A - AND
|	1	Logické NEBO - OR
^ nebo &&	1	Logické exklusivní NEBO - XOR

Vysvëtlení Booleovy algebry pro nematematiky

1.28 t-zvlznaky

3.1.5 Zvláîtní znaky

Nëkteré interpunkçní znaky mají v ARexxu speciální význam, jak následující tabulka objasñuje.

Zvláîtní znaky

Zvláîtní znaky	Definice
: Dvojteçka	Dvojteçka, pêed kterou stojí symbolový token (alfanumerický znak nebo .|?$), definuje skokovou značku v programu.
() Závorky	Závorky jsou pouùity ve výrazech ke spojení operátorû a operandû v podvýrazy a tím ruîí normální priority oněch operátorû. Otevírající závorka slouùí k označení volání funkce během výrazu.
Symbol nebo Êetëzec ,	po kterém bezprostêednë následuje otevírající závorka, definuje jméno funkce. V pokynu musí být stejný poçet otevírajících a zavírajících závorek.
; Stêedník	Stêedník slouùí jako koncový znak pokynu. Kdyù máte vícero krátkých pokynû, které se vejdou na êádku, dělí se od sebe stêedníkem.
, Çárka	Slouùí jako pokraçovací znaménko na konci êádky pro pokyn, rozdělený do několika êádkû, a jako oddělovaç mezi argumenty pêi volání funkce.

1.29 Vysvëtlení logických operátorû

B O O L E O V A A L G E B R A

Kaùdý programátor by sice měl matematiku dostateçnë zvládat, ale není to podmínkou. Proto pêedpokládám, ùe vám aspoň "struçné" vysvëtlení pêíjde vhod.

Obecný popis

Vëdní obor zabývající se pravidly, podle nichù se êeîí situace vnichù se jedná o jasné rozhodnutí na základë vstupních proměnných se nazývá binární logika. S takovými úkoly se potýkáme témëê kaùdou chvíli aniù bychom si to uvëdomovali. Tak napêíklad: Ráno vstanu,kouknu se do ledniçky, zjistím ùe je zcela prázdná, (logická proměnná) proto se rozhodnu zajít do obchodu... (logický výrok - výsledek logické operace)

Nejznámëjîím badatelem v tomto vëdním oboru byl George Boole [dùordù búl] Jeho jménem je označována soustava pravidel pro zápis a vyhodnocování logických vztahû. Êíká se ji Booleova algebra. Na první pohled to vypadá asi sloùitë, ale zdání klame. Poznat zásady Booleovy algebry je mnohem snaùîí, neù se nauçit algebru klasickou. Existuje v ní jen několik vztáhû, ale jednoduîîí je hlavnë vtom, ùe pouùívá pouze dvë çíslice,a to "1" a "0".

Tyto çíslice pêedstavují vstupní a výstupní promënné. Tëch vstupních mûùe být nekoneçnë mnoho a výstupní je vùdy jen jedna. Dëj, který musel proběhnou, aby tato hodnota, které se çasto êíká "výrok", vznikla se nazývá logická operace.

Poznámka: Çasto se mûùete setkat z jiným znaçením log. 1 nebo 0
0 -> nepravda -> FALSE -> LOW (v elektronice)
1 -> pravda -> TRUE -> HIGH (v elektronice)

Vztahy binární logiky

Existují têí základní, znichù se pak odvozují dalîí.

Logický souçet - NEBO - OR

Zápis: Y = A + B + C [çte se: Y se rovná A nebo B nebo C ("Nebo" = "OR")]
Popis: Výrok je pravdívý (Y=1) tehdy, jesliùe nëkterý z vstupních výrokû je pravdivý, tzn., ùe se rovná 1.
Pêíklad: If Exists(Soubor) | Pos(Text,string)>0 Then ...
- podmínka je splnëna, jestliùe existuje soubor nebo je pozice textu v êetëzci vëtîí neù 0.

Logický souçin - A,I - AND

Zápis: Y = A · B · C [çte se: Y se rovná A i B i C ("I" = "A" = "AND")]
Popis: Výrok je pravdivý tehdy, jestliùe jsou vîechny vstupní promënné rovny jedné. (A i B i C se musí rovnat 1.)
Pêíklad: If A=5 & EOF(VSTUP) & ... Then Exit 0
- program bude ukonçet, jestliùe se A=5, na vstupu se objevil koncový znak (EOF - End of file) a jsou splnëny vîechny dalîí podmínky.

Negace - NE - NOT

Zápis: $Y = \overline{A}$ [çte se: Y se rovná NE A, Y se rovná NON A]
Popis: Výsledek operace je opaçný vstupní promënné. (1 -> 0, 0 -> 1)
Pêíklad: If ~Open(OUT,výstup) Then Say 'Otevêení se nezdaêilo!'
- Pokud se nepodaêí otevêít výstup, vrátí funkce "0" a ta je následnë negována na "1". Podmínka je splnëna, zpráva vypsána.

Kombinací se tvoêí:

Exklusivní (Vyjímeçné) NEBO (OR) - XOR

Zápis: $Y = A \cdot \overline{B} + \overline{A} \cdot B$ [çte se: Y se rovná A i NE B nebo NE A i B]
Popis: Výrok je pravdivý jen tehdy, jestliùe je pravdivá právë jedna vstupní promënná. (Viz. pravdivostní tabulka)
Pravdivostní tabulka:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Dalîí vznikají negací základních operací, znací se vùdy pêidáním çáry nad pravou stranu, aby bylo zêetelné, ùe výsledek je nejprve obrácen.

```
    NOR  - negovaný (pêevrácený) výsledek OR (NEBO)
    NAND - negovaná log. funkce AND (a,i)

    YES  - znovu negovaná negace ( vstup = výstupu, Y = A)
```

Celkem zajimavý vědní obor, ùe? Jiù teð toho víte víc neù budete pravdëpodobnë potêebovat. Pro zajímavost existují têeba jeîte De Morganovy zákony, ale to bychom pêíliî zacházeli od naîí tématiky. To je záleùitost hlavnë matematikû a elektrotechnikû. Pokud to budete chtít vëdët navîtivte knihovnu.

Pro vás, jako programátory bude uùiteçnëjîí, kdyù si blíùe popíîeme

tvorbu výrazû v ARexxu
.

1.30 Jak vyzrát na tvorbu výrazu a podmínek...

Popis tvorby výrazû pro tvorbu podmínek

Následující popis se zabývá tvorbou pêedevîím logických podmínek jejichù výsledkem je
booleovská hodnota
TRUE (1) nebo FALSE (0). Nëkteré záleùitosti se vîak týkají vîech výrazû.
Obecným popisem
se zabývá jedna
z úvodních kapitol, takùe základ by jste jiù mëli mít za sebou.

Interpet ARexxu êídí vyhodnocování výrazû pomoci priorit. Çím vyîîí prioritu má daný operátor (napê +), tím je jeho vyhodnocení upêednostnëno. Pokud potêebujete zvíîit prioriu nëjakého operandu slouùí k tomu závorky.

Jako pêíklad pouùiji napêíklad tento výraz: 10/5*3-1 = 6 | 2 = 7%3. arexxový interpet nejdêíve zpracuje násobení, dëlení a celoçíselné dëlení, protoùe vîe má prioritu 6. Výraz pak bude vypadat takto: 6-1 = 6 | 2 = 2.
Dále bude provedeno odçítání, to má prioritu 5. Následnë pak bude vyhodnocena rovnost, protoùe ta má prioritu 3.
Vznikne tak tento výraz: 0 | 1. Jak na to pêiîel ?
Jednoduîe se zeptal: 5=6 ? Jelikoù tento výrok je nepravdivý, jedná se o leù a té
booleova algebra
pêidëluje çíslo 0, zatímco pravdë 1.
Ke klasické matematice se nevrátíme ani nyní. Opët se budeme êídit pouze zákonitostmi booleovy algebry. Znak "|" pêedstavuje logický opreátor NEBO (OR), tzn., ùe "výraz je pravdivý, pokud je pravdivý jeden ze vstupních výrokû. A protoùe jeden pravdivý je (2=2), je celý tento výraz pravdivý.

Pamatujte si, ùe
booleova algebra
pracuuje jen s çísly 1 a 0, nesmí se
tedy nikdy stát, ùe interpetu ARexxu pêedloùíte tento výraz: 5 | 6, protoùe to by znamenalo chybu ç.

```
                46
                :Boolean value not 0 or 1.
```

Nyní se podíváme na pêíklad, který je více z praxe:

```
                IF

                Exists
               ('t:soubor') & zprava~=1 Then
                Say
                'Soubor existuje'
```

Zde jsou pouùity také funkce, pokud na në interpet narazí pozastaví analýzu výrazu, zpracuje funkci a do výrazu místo ní dosadí výsledek, který funkce vrátila. Vîechny funkce jsou vyhodnoceny pêednostnë.

Pokud tedy existuje soubor "t:soubor", vrátí funkce 1. Následnë záleùí na tom, zda se promënná "zpráva "rovná 1 nebo ne. Nás vlastnë zajímá zda se promënná nerovná 1, protoùe "~=" je operátor pro 'nerovnost'.

Dalîím krokem je vyhodnocení logického souçinu "&". Ten se rovná jedné, jen tehdy, jestliùe se vîechny vstupní výrazy rovnají 1.

Z pêedchozího je patrné, ùe text bude na obrazovku vypsán jen v tom pêípadë, kdy existuje soubor "t:soubor" a promënná "zpráva" je rûzná od 1.

Takovýmto zpûsobem mûùete skládat libovolnë sloùité výrazy. Je zde vîak jedno omezení, výraz mûùe mít maximálnë 32 úrovní, protoùe jinak dojde k pêeplnění vnitêního zásobníku, coù má za následek vznik chyby

```
                43
                : Expression nesting >32.
```

Kolik je 32 úrovní a coje to? Pêed chvílí jsme si vysvëtlili, jak probíhá postupné zjednosuîování výrazu aù k booleovské hodnotë (pokud se jednalo o logický výraz) nebo k nëjakému çíslu (pokud je jednalo o matematický výraz). A tëchto krokû mûùe být maximálnë 32. Êeîit tento problém lze velice jednoduîe, staçí výraz rozdëlit na nëkolik mënîích a jednoduîîích.

Pokud máte pocit, ùe jste této çásti nerozumëli, zkuste znovu prostudovat çást zabývající se Operátory, která je souçástí popisu Tokenû.

Nebo rovnou prostudujte celou têetí kapitolu Prvky ARexxu.

1.31 Klausule, Dodatky

```
                3.2  Klausule
 -------------
```

Klausule (dodatek) se skládá ze skupin tokenû a pêedstavuje nejmenîí, jako pokyn proveditelnou jednotku.

Pêi çtení programu interpret rozdëlí program na skupiny klausulí. Tyto skupiny se dále dëlí na tokeny, a kaùdá klausule je potom urçitým typem klasifikována. Pêitom mohou nezêetelné syntaktické rozdíly sémantický obsah pokynu zcela zmënit. Pêíklad: SAY 'Amiga=Multimedia' je pêíkazová klausule a ukáùe "Amiga=Multimedia" na obrazovce, ale: 'SAY' 'Amiga=Multimedia' je kommandová klausule a pêedá "Amiga=Multimedia" na externí program. Vedoucí ('') êetëzec pûsobí na klasifikování: z pêíkazové klausule je kommandová klausule.

Konec êádku obvykle platí jako implicitní konec klausule. Klausule mûùe ale pokraçovat na dalîím êádku. Kvûli tomu musí být první êádek ukonçen çárkou. çárka bude programem ignorována a dalîí êádka bude interpretována jako pokraçování klausule. Pro takováto pokraçování není pevné maximum (vhledem k velikosti kommandového bufferu (zásobníku)).

Êetëzcové a komentáêové tokeny pokraçují automaticky po dosaùení konce êádku, pokud jeîtë nebylo dosaùeno hraniçního znaku. Znak pro nový êádek (který generuje ENTER) není povaùován za souçást tokenu.

Pêíklad:
```
 /* Ukázka êetëzce pêes více êádkû - TP */
 prom = '
 první êádek
 druhý êádek
 poslední êádek'

 Say prom          ->  první êádekdruhý êádekposlední êádek
```

3.2.1 Nulové klausule

Êádky, skládající se výluçnë z prázdných znakû nebo komentáêû, jsou povaùovány za nulové klausule. Mûùou se vyskytnout na jakémkoliv místë v programu. Nemají význam pro program. Zvyîují pouze pêehlednost programu a poçet êádkû (a velikost programu).

3.2.2 Skokové klausule

Symbol, po kterém následuje dvojteçka (:), se oznaçuje jako skoková klausule. Skoková znaçka slouùí jako oznaçovací znamení v programu. Provedení skokové znaçky nezpûsobí ùádnou akci. Dvojteçka je povaùována za implicitní zakonçení klausule, to znamená, ùe kaùdá skoková znaçka tvoêí separátní klausuli. Skokové klausule se mohou vyskytovat na kaùdém libovolném místë v programu.

Mûùete tak definovat libovolné návëîtí, na které pak mûùete kdykoliv skoçit pêíkazem "CALL" a zadáním názvu návëstí. (Bez dvojteçky.)

3.2.3 Pêiêazovací klausule

Pêiêazovací klausule je oznaçena promënným symbolem ,po kterém následuje operátor = (v tomto kontextu je normální definice operátoru = (zkoumání rovnosti) zruîena). Tokeny vpravo od rovnítka jsou vyhodnoceny jako výraz, a výsledek je pêiêazen promënnému symbolu.

Pêíklad:

```
when = 'Uù je naçase'
answ = 3.14 * fact(5)
```

Rovnítko = pêiêadí hodnotu 'Uù je naçase' promënné 'when' a výsledek poçetní operace 3.14 * fact(5) promënné 'answ'.

3.2.4 Pêíkazové klausule

Pêíkazové klausule zaçínají jménem pêíkazu a dávají ARexxu pokyn k provedení akce. Pêíkazy jsou popsány v kapitole 4. Pêíklad:

```
DROP a b c
SAY 'Prosím'
IF j > 5 THEN LEAVE;
```

3.2.5 Kommandové klausule

Kommandové klausule jsou vîechny arexxové výrazy, které nespadají do uvedených kategorií. Výraz je vyhodnocen, a výsledek je pêedán na externí host (hostitelský program). Externí host lze zvolit pêíkazem ADDRESS LINK ADDRESS}.

Pêíklad:

```
'delete' 'Data'   /*AmigaDOSový pêíkaz*/
'jump' current+10 /*Editorový pêíkaz*/
```

Pêíkaz "delete" nebude jako arexxový pêíkaz rozpoznán a proto vyslán na externí host - v tomto pêípadë AmigaDOS. Pêíkazu "jump" ve druhém pêíkladu by porozumël textový editor.

1.32 Vyrazy

3.3 Výrazy

Pod výrazy se rozumí skupina vyhodnocených tokenû. Çasto pokyny obsahují více neù jeden výraz. Výrazy se skládají z následujících komponentû (sloùek):

* Êetëzec – hodnota êetëzce je êetëzec sám. Êetëzce se zapisují do uvozovek (") nebo apostrofû (’). Maximální délka êetëzce je 65 535 znakû.

* Symboly – hodnota pevného symbolu je symbol sám, pêemënën na velká písmena.
Symboly
mohou být pouùity jako promënné, mohou tedy mít pêiêazenu hodnotu.

* Operátory – operátory mají prioritu, která urçuje poêadí pêi provádëní operací.

* Závorky – Kulaté závorky mohou být pouùity na zmënu poêadí pêi vyhodnocování výrazu, nebo oznaçit volání funkce. Symbol nebo êetëzec, po kterém bezprostêednë následuje otevêená závorka, definuje jméno funkce, tokeny mezi závorkami tvoêí seznam argumentû. Takùe následující výraz se skládá z:

J ’Fakulta je’ fact(J)

- jeden symbol – J
- jeden prázdný operátor
- jeden êetëzec – ’Fakulta je’
- dalîí prázdný znak
- symbol – fact
- otevêená závorka
- symbol – J
- zavírající závorka

V tomto pêíkladu je FACT jméno funkce a (J) k tomu patêící seznam argumentû, který v tomto pêípadë obsahuje jen samotný výraz J.

Pêed vyhodnocením výrazu potêebuje ARexx hodnotu pro kaùdý
symbol
ve výrazu. Pêi pevných symbolech je hodnotou symbolu sám ↩
symbol, pro
promënné symboly se musí ARexx podívat do aktuální symbolové tabulky. V horním pêíkladu by výraz vypadal asi takhle za pêedpokladu, ùe J by byla pêiêazena hodnota 3:

3 ’Fakulta je’ FACT(3)

Aby se vyvaroval nejasností, co se týçe hodnot, pêiêazených symbolûm bëhem procesu rozkladu, zaruçuje ARexx striktní dodrùení rozkladového poêadí zleva doprava (poêadí, jak prochází programem êádku po êádce). Rozklad symbolû se dëje nezávisle na prioritë operátorû a postavení závorek. Je-li zjiîtëno volání funkce, je rozklad bëhem vyhodnocení funkce pozastaven. Dejte si pozor, neboï jeden a tentýù symbol mûùe mít více neù jednu hodnotu.

Vezmëme si horní pêíklad následovnë pêeskupen:

```
FACT(J) 'je' J 'Fakulta'
```

Bude druhé J rozloùeno na hodnotu 3? Vîeobecnë mohou mít volání funkcí vedlejîí efekty, napê. zmënu hodnot promënných. Volání FACT() v pêeskupeném výrazu by mohlo zmënit hodnotu J, ùe pêi pêíîtím výskytu J by jiù byla úçinná nová hodnota.

Po rozkladu vîech hodnot symbolû je výraz vyhodnocen podle priorit operátorû a pozic závorek. Pêi vyhodnocování operátorû se shodnou prioritou nezaruçuje ARexx ùádné urçité poêadí.

Dále nepouùívá ARexx pêi vyhodnocování booleovských operací ùádné "zkratky". Pêi vyhodnocování výrazu: (1 = 2) & (FACT(3) = 6) bude zavolána funkce FACT() i kdyù první çlen operace je 0. Na tomto pêípadë je zêejmé, ùe ARexx pokraçuje se çtením zleva doprava, i kdyù první závorka v pêíkladu je logicky chybná a dává hodnotu 0 a tím je jasné, ùe koneçný výsledek bude také 0.

Poznámka: Jiù víte,ùe "&" znaçí logický souçin a ten se vyznaçuje tím, ùe výsledek je pravdivý jen tehdy, jso-li pravdivé vîechny vstupní hodnoty. Je tedy zbyteçné pokraçovat dále,kdyù víme,ùe 1 se nikdy nerovná 2, tudíù tento výrok je nepravdivý. (Y= 0 · 1 -> Y=0). ARexx vîak nejprve vîe rozloùí a teprve potom vyhodnocuje zda podmínka platí nebo nikoli...

Více se dozvíte v çásti popisující tvorby výrazû pro tvorbu podmínek

1.33 Kommandové rozhraní

```
                3.4  Kommandové rozhraní
 ------------------------
```

Arexxové kommandové rozhraní je vîeobecnë pêístupný Message-port. S ARexxem kompatibilní aplikace musí mít Message-port. Arexxové programy posílají kommanda sbalením kommandového êetëzce do Message-paketu (balíku) a tento paket poîlou na Message-port hostu (hostitelský program - host). Program vysadí, dokud host zpracovává kommanda. Kdyù se zprávový paket (balík) pêihlásí zpátky, pokraçuje program v práci.

3.4.1 Adresa hostu

```
-------------------
```

ARexx spravuje dvë implicitní adresy hostu (host - hostitelský program) - aktuální a pêedchozí hodnotu - jako souçást paměïového prostêedí programu. Tyto hodnoty mohou být kdykoliv zmëněny pêíkazem ADDRESS (nebo synonymem

SHELL). Aktuální adresa hostu mûùe být naçtena integrovanou funkcí

ADDRESS()

. Standardní êetëzec pro adresu hostu je REXX, to ale mûùe být zmëněno pêi volání programu. Vëtîina hostových aplikací poskytuje jméno jejich vîeobecnë pêístupného portu. Kdyù zavoláte makro program, makro mûùe automaticky kommanda pêedat na host.

Speciální adresa hostu je rozpoznávána. êetëzec COMMAND udává, ùe makro má být pêedáno pêímo na AmigaDOS. U ostatních hostových adres se vychází z toho, ùe se vztahují na veêejný Message-port. Pokus o zaslání zprávy na neexistující Message-port vede k syntaxní chybë.

Následující program ukazuje interakci mezi ARexxem a AmigaDOSovým editorem, ED. Program zjiîïuje,zda ED bëùí, urçuje jméno Message-portu a zêizuje nëkteré kmenové promënné.

Program 9. InterakceED.rexx Spusï program !

```
! Poznámka !
/* Vyhodí status EDu. ED musí bëùet pêed spuîtëním tohoto programu ↩
   . Ed-ové
porty se jmenují 'Ed', 'Ed_1', 'Ed_2' atd. */
DEFAULT_ED = "Ed"   /* Rozliîování velkých písmen je zde relevantní*/
/* Procedura pro pêípad, ùe ED nebëùí */
DO WHILE ~ SHOW('p',DEFAULT_ED)           /* Hledání portu */
   SAY "Nenalezen: Port se jménem" DEFAULT_ED
   SAY "Pouùitelné porty:"
   SAY SHOW('P',,',')
   SAY "Zadejte jiné jméno portu nebo QUIT k opuîtëní
   programu"
   /* Uùivatel má zvolit port, pokud jej program nenajde */
   DEFAULT_ED = READLN(
                STDIN
               )     /* Lze pouùít také "
                PULL
               " */
   IF STRIP(UPPER(DEFAULT_ED)) = 'QUIT' THEN EXIT 10
   /* Tímhle mûùe uùivatel opustit program */
END
SAY "Pouùíván bude port" DEFAULT_ED
/* Kdyù uù je port koneçnë nalezen, mûùe ho ARexx adresovat */
ADDRESS VALUE DEFAULT_ED
/* Pár potêebných kmenových promënných je zêízeno */
STEM.0 = 15           /* Poçet Ed-arexxových promënných */
STEM.1 = 'LEFT'       /* Levý roh (SL)*/
STEM.2 = 'RIGHT'      /* Pravý roh (SR)*/
STEM.3 = 'TABSTOP'    /* Tabulátory (ST)*/
STEM.4 = 'LMAX'       /* Maximálnë zobrazitelné êádky */
STEM.5 = 'WIDTH'      /* ííêka displaye ve znacích */
STEM.6 = 'X'          /* Fyzická X-ová posice, od 1 */
STEM.7 = 'Y'          /* Fyzická Y-ová posice, od 1 */
STEM.8 = 'BASE'       /* Základna okna */
  /* Základna je 0, pokud není zpráva pêesunuta doprava */
STEM.9 = 'EXTEND'     /* Rozîíêit hodnotu okraje (EX) */
STEM.10 = 'FORECASE'    /* Psaní velkých a malých písmen */
STEM.11 = 'LINE'      /* Aktuální poçet êádkû */
STEM.12 = 'FILENAME'    /* Soubor bude editován */
```

```
STEM.13 = 'CURRENT'      /* Text aktuální êádky */
STEM.14 = 'LASTCMD'      /* Poslední rozîíêené kommando */
STEM.15 = 'SEARCH'       /* Poslední hledaný êetëzec */
/* Ed má hodnoty do kmenové promënné 'STEM.' vloùit. */
'RV' '/STEM/'      /* RV je Ed-ové kommando k posílání informací*/
/* STEM.1 je LEFT, a STEM.LEFT má nyní hodnotu z ED-u. Tak mohou být tato
data vypsána */

DO i = 1 to STEM.0
   ED_VAR = STEM.i
   SAY STEM.i "=" STEM.ED_VAR
   /*Ed-ové promënné a hodnoty vypsat*/
END
```

3.4.2 Tvorba makra

ARexx mûùe být pouùit k psaní programû pro kaùdou host-aplikaci, která vlastní kompatibilní kommandové rozhraní. Nëkteré aplikaçní programy jsou psány na bázi integrovaného makro-jazyka a mohou nabídnout poçetné makro-kommanda.

Pêezkouîejte aplikaçní program na "shortcut" kommanda (klávesové zkratky). Nëkteré programy poskytují výkonné funkce, které byly speciálnë implementovány pro volání makro-programû (jiný výraz pro arexxové programy).

Interpretace pêijatých kommand závisí výluçnë na dané host-aplikaci. V nejjednoduîîím pêípadë odpovídá kommandový êetëzec exaktnë kommandûm, které mohou být zadány uùivatelem. Kommanda, êídící pozici v textovém editoru, budou pravdëpodobnë stejnë interpretována. Dalîí kommanda jsou ale jen platná, kdyù jsou zadána z makra. Kommando k simulaci operace v menu by asi bylo tëùko zadáno uùivatelem. V následujícím pêíkladu je arexxový program vyvolán z ED-u k záměnë dvou znakû.

Program 10. Transpose.rexx Spusï program !

```
                ! Poznámka !
              Poznámka: Pêi spuîtëní programu je o vîe postaráno ! Program, ↩
                 který se
          spouîtí  je totiù rozîíêen o spuîtëní  EDu a vepsání çíslic.

/* Daný êetëzec '123', kdyù je kursor na 3, makro pêemëní na '132' */
HOST = ADDRESS()       /* Z kterého Ed-u pêiîlo volání ? */
ADDRESS VALUE HOST     /* . . . s editorem komunikovat */
'RV' '/CURR/'        /* Ed má informaci vloùit do kmene CURR */

/*Potêebujeme dva údaje:*/
currpos = CURR.X       /* Kursorová pozice */
currlin = CURR.CURRENT
/* Obsah aktuální êádky */

IF (currpos > 2)       /* Minimálnë 3. místo */
    THEN currpos = currpos - 1
    ELSE DO          /* Ohlásit chybu a opustit program */
    Say 'Cursor musí stát minimálnë na têetí pozici!'
    EXIT 10
```

```
END

/* Znaky CURRPOS a CURRPOS-1 zamënit a aktuální êádek nahradit novým */
DROP CURR.  /*SYSTEM-ová proměnná CURR není jiù potêeba; îetêíme pamëtí. */

'd'             /* Ed má smazat aktuální êádek */
currlin = swapch(currpos,currlin)   /* Dva znaky vymënit */
'i /'||currlin||'/'         /* Změněný êádek pêipojit */
DO i = 1 to currpos         /* Kursor zpët na staré místo */
   'cr'           /* Ed-ový pêíkaz - kursor doprava*/
End

EXIT            /* Vîe hotovo */

/* Funkce na výměnu dvou znakû */
swapch: procedure
PARSE ARG cpos,clin
    cl = substr(clin, cpos, 1)    /* naçíst znaky */
    clin = delstr(clin, cpos, 1)    /* vymazat ze êetëzce */
    clin = insert(cl,clin,cpos-2,1)   /* Pêidat, aby transpozice byla
                                         kompletní */
RETURN clin         /* Změněný êetëzec vrátit */
```

K zavolání tohoto pêíkladu z Ed-u zmáçknëte prosím ESC (v ED-u) a zadejte následující:

```
RX "Rexx:transpose.rexx"
```

Zde musí být zadána plná cesta a koncovka (.rexx). Tento êetëzec mûùe být pêirozenë pêiêazen funkçní klávese.

3.4.3 Návratové kódy (return-codes)

Po uzavêení zpracování kommand ohlásí host-kommando stav ve formë návratového kódu. Vîechny moùné návratové kódy najdete ve své dokumentaci k Vaîí host-ové aplikaci. Tyto kódy ukazují, zda kommandem provedená aplikace byla úspëîná.

Tento návratový kód je uloùen do speciální arexxové proměnné RC, aby mohl být makrem pêezkouîen. Hodnota nula urçuje úspëîné provedení kommanda. Pozitivní celé çíslo naznaçuje moùnou chybu. çím vyîîí çíslo, tím vëtîí chyba. Z návratového kódu mûùe makro-program urçit, zda kommando bylo provedeno, nebo v pêípadë selhání, zda je zapotêebí provést akci.

3.4.4 Kommandové Shelly

ARexx je sice postaven tak, ùe s programy, které podporují jeho specifické kommandové rozhraní, lze pracovat nejefektivnëji, ale v zásadë je moùno jej pouùít s kaùdým shellovým (CLI) programem, který podporuje standardní I/O mechanismy k vyvolání svého datového proudu. Jedna moùnost spoçívá ve vyuùití ARexxu k vytvoêit pêíkazový soubor na disku a tento soubor dále vést na Shell. V programu 11 je otevêen nový Shell k provedení standardního EXECUTE skriptu:

```
 Program 11.  Shell.rexx        Spusï program !
                ! Poznámka !
               /* Spustit nový Shell*/

conwindow = "CON:0/0/640/100/NovýShell/Close"

/*Vytvoêit pêíkazový soubor*/
CALL OPEN('OUT',"ram:temp",'WRITE')
CALL WRITELN('OUT', 'echo "Toto je test"')
CALL CLOSE('OUT')

/*Otevêít Newshellové okno*/
ADDRESS command 'newshell' conwindow 'ram:temp'
EXIT

Poznámaka: Jakýkoliv  pêíkaz (program)  který lze  spustit  z Shellu mûùete
           v ARexxu pouùít prostêednictvím pêíkazu
                ADDRESS
                COMMAND.
```

1.34 Provozní prostêedí

```
                3.5  Provozní prostêedí
 ------------------------
```

Následující informace se zamëêují na zkuîené uùivatele Amig. Pêedpokládají základní znalosti systému Amigy a znalost "Amiga Rom Kernel manuálu".

Arexxový interpret
RexxMast
nabízí jednotné provozní prostêedí, tím, ùe kaùdý program provádí jako separátní proces v multitaskovém prostêedí Amigy. Tím získáváte pruùné (flexibilní) rozhraní mezi externím host-programem a RexxMast-em. Host-program mûùe provádët své operace souçasnë nebo mûùe çekat, aù je interpretovaný arexxový program ukonçen. Kaùdý arexxový program má vnitêní a vnëjîí prostêedí.

3.5.1 Externí prostêedí

K externímu prostêedí programu patêí procesové struktury, vstupní a výstupní
datové proudy
, jako i aktuální adresáê. Pêi zêízení kaùdého arexxového procesu pêebírá tento proces jak vstupní a výstupní proudy svého klienta, tak i externího programu , který zavolal arexxový program. Byl-li napê. arexxový program odstartován ze Shellu, pêebírá arexxový program vstupní a výstupní proudy jako i aktuální adresáê onoho Shellu. Hledání programu nebo dat vychází pak z aktuálního adresáêe.

3.5.2 Interní prostêedí

Interní prostêedí arexxového programu sestává z jedné statické globální struktury a jednoho nebo více pamëïových prostêedí. Globální datové hodnoty jsou v okamùiku volání programu pevné hodnoty (statické). K tëmto hodnotám patêí zdrojový text programu, statické datové êetëzce a argumentové êetëzce. Jakmile program bëùí, nemohou být tyto hodnoty zmëněny.

arexxové programy, které jsou volány jako kommanda, mají bëùnë jen jeden argumentový êetëzec, i kdyù moùnost dëlení kommand (tokenizace) nabízí moùnost pouùití vícerých argumentových êetëzcû. Program, volaný jako interní funkce, mûùe mít libovolný poçet argumentû. Tyto argumenty bëhem prûbëhu programu zûstávají.

Provoz. prostêedí zahrnuje
symbolovou
tabulku,pouùívanou pro proměnné,
çíselné volby, monitorování a host-adresové êetëzce. Bëhem vykonávání programu mûùe být jen jedno globální prostêedí, ale více pamëïových prostêedí. Pêi kaùdém volání
interní funkce
je aktivováno nové pamëïové
prostêedí a inicializováno. Výchozí hodnoty pro vëtîinu polí jsou vzaty z pêedchozího prostêedí. Hodnoty mohou být pozdëji zmëněny, aniù by se to odrazilo na prostêedí volající úrovnë (úroveň, odkud pochází volání). Nové prostêedí zûstává tak dlouho, dokud funkce nevrátí êízení.

Ke kaùdému pamëïovému prostêedí patêí
symbolová
tabulka. Slouùí k
ukládání hodnotových êetëzcû, které jsou pêiêazeny proměnným. Tato symbolová tabulka má formu jednoho dvouúrovňového binárního stromu. Do primární úrovnë se ukládají zápisy pro jednoduché symboly a kmenové symboly, do sekundární pro sloùené symboly. Vîechny sloùené symboly se stejným kmenem jsou zapsány do stromu, kde zápis kmenu tvoêí koêen stromu.

Symboly jsou vzaty do tabulky teprve po probëhlém pêidëlení. Na primární úrovni vytvoêené zápisy nejsou nikdy smazány, i kdyù je pozdëji vykonána zpëtná inicializace symbolu. Sekundární stromy jsou uvolněny, kdyù je vykonáno pêidëlení na kmen, patêící tomuto stromu.

3.5.3 Správa provozních prostêedkû

ARexx nabízí kompletní správu vîech dynamicky pêiêazených provozních prostêedkû (resource tracking), pouùitých k provedení programu. K tëmto resource patêí místo v pamëti, DOSové soubory a pêíbuzné struktury jako i struktura Messsage-portu. Tento systém správy je tak vyváùen, ùe pêi pêeruîení programu v libovolném okamùiku nezûstanou ùádné provozní prostêedky blokovány.

Skrz pêímé volání systému arexxovým programem je moùné systém správy provozních prostêedkû interpretu obejít. V tomto pêípadë je úloha programátora, vîechny provozní prostêedky mimo arexxový systém správy prostêedkû kontrolovat. ARexx nabízí speciální systém pêeruîení, takový,

ùe si program udrùí êízení i pêi chybë ve vykonávání programu, vykoná potêebné vyçiîtëní a podle pravidel mûùe být ukonçen.

1.35 4. kapitola

Pêíkazy

Pêíkazová klausule zaçíná jménem urçitého pêíkazu a navádí ARexx k provedení urçité akce. Tato kapitola obsahuje seznam v ARexxu dostupných pêíkazû.

Po kaùdém pêíkazovém klíçovém slovu mûùe následovat jeden nebo více pod-klíçových slov, výrazy, nebo jiné, pro pêíkaz typické informace. Pêíkazová klíçová slova a pod-klíçová slova jsou rozpoznány jen v tomto specifickém kontextu. Proto je moùné pouùít stejná klíçová slova v rûzných kontextech jako názvy proměnných nebo funkçní jména. Po pêíkazovém klíçovém slovu nesmí následovat ùádný z operátorû (:) nebo (=).

Poznámka: Pêed nëkterými pêíkazy se vyskutuje znak, který znamená následující: * - sekundární pêíkaz - nelze pouùít samostatnë, protoùe ukonçuje oblast definovanou jiným pêíkazem, napê. DO -> END

4.2 Abecední rejstêík pêíkazû ARexxu

Abecední seznam	Syntaxe
A	
ADDRESS	- [[<název portu> [<pêíkazový výraz>]] | [[VALUE] Výraz]]
ARG	- < îablona >
B	
BREAK	C
CALL	- <název> [<výraz>][,<výraz> ...]
D	
DO	- [[<proměnná>=<výraz>] [TO <výraz>] [BY <výraz>]] [FOR <výraz>] [FOREVER] [WHILE < výraz > | UNTIL <

```
                výraz
               >]

                DROP
                - <proměnná> [<proměnná> ...]

E
-------------------------

                ECHO
                - [<výraz>]

                * ELSE
                - [;] [<alternativní pokyn>]   (Souçást
                IF
               ,
                Select
               )

                * END
                - [<jméno cyklu>]            (Souçást
                DO
               ,
                Select
               )

                EXIT
                - [<výraz>]

I
-------------------------

                IF
                - <
                výraz
               > [;] THEN [;] [<podmíněný pokyn>] [; ELSE [;] <alternativní pokyn ↩
                  >]

                INTERPRET
                - <výraz>

                ITERATE
                - [<jméno>]

L
-------------------------

                LEAVE
                - [<jméno>]

N
-------------------------

                NOP

                NUMERIC
```

```
                - {DIGITS | FUZZ} <výraz> nebo FORM {SCIENTIFIC | ENGINEERING | [ ↩
                   VALUE] <výraz>}

O
-------------------------

                OPTIONS
                - {FAILAT <výraz>}|{PROMPT <výraz>}|{RESULTS}|{CACHE}

                * OTHERWISE
                - [;] [<alternativní pokyn>]  (Souçást
                SELECT
               )

P
-------------------------

                PARSE
                - [UPPER] Zdroj [<
                îablona
               >]

                PROCEDURE
                - [ EXPOSE <promënná> [<promënná>] [...] ]

                PULL
                - [<
                îablona
               >]

                PUSH
                - [<výraz>]

Q
-------------------------

                QUEUE
                - [<výraz>]

R
-------------------------

                RETURN
                - [<výraz>]

S
-------------------------

                SAY
                - [<výraz>]

                SELECT

                SHELL
                - [ [<název portu> [<pêíkazový výraz>]] | [[VALUE] Výraz] ]

                SIGNAL
```

```
                - {ON | OFF} <interrupt> nebo ë[VALUE] <název návëstí>

 T
 -------------------------

                TRACE
                - {[[{!|?}] [volba]]} | {[VALUE <výraz>]} | [-<çíslo>]

 W
 ------------------------

                * WHEN
                - <
                výraz
               > [;] THEN [;] [<podmíněný pokyn>]  (Souçást
                SELECT
               )
```

1.36 Popis pêíkazu ADDRESS

```
                          | <jméno portu>  [<pêíkazový výraz>]
 ADDRESS | COMMAND [<pêíkazový výraz>]
         | [VALUE] <výraz>
         | (bez argumentu)
```

Tento pêíkaz slouùí pro práci s host-adresami. Pod host-adresou se rozumí jméno Message-portu aplikace, na který jsou posílány ARexxová kommanda. ARexx spravuje dvë host-adresy, a to aktuální a pêedchozí. Vùdy, kdyù zadáte novou adresu, tak se automaticky stane dosud aktuální adresa adresou pêedchozí. Pêiçemù dosud pêedchozí adresa je smazána ! Host-adresa je souçástí paměïového prostêedí programu a pêetrvává i vnitêní volání funkcí.

Aktuální adresa mûùe být být vyvolána integrovanou funkcí ADDRESS().

Pêíkaz ADDRESS - zadán bez argumentû - změní pêedchozí hodnotu na aktuální a aktuální za pêedchozí. Pêi opakovaném pouùití se opët pêehodí adresy mezi sebou.

ADDRESS <jméno portu> dëlá ze zadaného êetëzce nebo symbolu novou adresu. Hodnota êetëzce nebo symbolu pêedstavuje název portu. Pêi jménech Message-portû se rozliîuje psaní velkých a malých písmen! Programové kommando na Message-port "Muj_port" by muselo vypadat takhle:

```
ADDRESS 'Muj_port'
```

Kdyù vynecháte jednoduché uvozovky, hledá ARexx Message-port MUJ_PORT a vyhodí chybovou hláîku, proto je dûleùité dodrùovat velikost písmen.

Pokud po êetëzci nebo symbolu následující nëjaký výraz, bude vyhodnocen a výsledek bude poslán na zadaný zadaný port. Aktuální a pêedchozí adresa se ale nezmëní. Tímto zpûsobem je moùné dát kommando na externí host, aniù by se zmëníly host adresy. S návratovou hodnotou kommanda je zacházeno jako s kommandovou klausulí (Promënná "Result").
Pêi pêedávání pêíkazu externímu hostu je vhodné uzavêít pêedávaný êetëzec do apostrofû nebo uvozovek, zvláîtë jedná-li se více neù o jedno slovo. V následujícím pêíkladë je portu 'rexx_ced' pêedán êetëzec 'TEXT ahoj', který je vîak jeîtë pêed odesláním doplnën o hodnotu promënné "A". Je-li hodnota "A" napê. 'kámo', bude na port posláno 'TEXT ahoj kámo'. Tento êetëzec zpracuje program zpravující daný port jako pêíkaz 'TEXT' s prarameterm 'ahoj kámo'.

Pêíklad: ADDRESS 'rexx_ced' 'TEXT ahoj' A

Pêi zadání ADDRESS [VALUE] <výraz> puùije ARexx výsledek výrazu jako novou host adresu (doteðka aktuální adresa bude pêedchozí). Klíçové slovo VALUE mûùe být vynecháno, potom musí být celý výraz v závorce.

Pêíklad:

```
ADDRESS               /* Výmëna aktuální adresy za pêedchozí */
ADDRESS 'rexx_ced'        /* Nová host adresa se jmenuje EDIT    */
ADDRESS 'rexx ced' 'TEXT Ahoj'  /* Vypíše text do CEDu                 */
ADDRESS VALUE spocti()  \_  /* Spoçítat novou host-adresu pomoci   */
ADDRESS( spocti() )     /       /*
                interní funkce
               .                   */
```

Podívejte se také na:
ADDRESS()
jako funkci.

1.37 Popis pêíkazu ARG

```
ARG <
îablona
>
```

ARG je zkrácená forma pêíkazu
PARSE
UPPER ARG.

ARG slouùí k pêevzetí êetëzce z argumenty, jeho rozkladu a pêídëlení jednotlivým promënným. "êetëzec s argumenty" vzniká voláte-li arexxový program nebo
interní funkci
parametry.

Argumentové êetëzce se pêíkazem ARG nemëní.
ARG pêevádí pêed rozkladem êetëzec na velká písmena.

```
  Viz:
                Provozní prostêedí
                Pêíklad:

 Say Secti(1,2)

 Exit

 Secti: Procedure
  ARG A,B                                  /*Vyvolání argumentû*/
  Say 'Sçítám' A '+' B '...'
 Return A+B

   Struktura  a  zpracování  îablon  je popsáno v oddíle zabývajícím se

                Syntaxní analýzou
               .

Podívejte se také na:
                PARSE ARG
```

1.38 Popis pêíkazu BREAK

```
                BREAK

   Pêíkaz  BREAK  slouùí k pêedçasnému  opuîtëní  oblast  pêíkazu
                DO
                nebo

                INTERPRET
               ovaného êetëzce.  Jen jen v této souvyslosti je pouùitelný.

   Je-li   pouùit   uvnitê   nëkolika  DO  pokynû,  je  opuîtën  pouze  ten
nejvnitênëjîí, který obsahuje BREAK.

   Tomuto pêíkazu je podobný pêíkaz
                LEAVE
               ,s kterým vîak lze opustit pouze
iterativní (opakující se) DO smyçku.

 Pêíklad:

DO          /*Zaçít blok*/
    IF a>3 THEN BREAK   /*Hotovo? Pokud ano, program skoçí aù za END*/
    a = a + 1
    y.a = name
END

 Podívejte se také na:
                DO

                LEAVE
```

```
                ITERATE
```

1.39 Popis pêíkazu CALL

```
                CALL <název> [<výraz>][,<výraz> ...]

   Pêíkaz CALL slouùí k zavolání
                interní
                nebo
                externí

                funkce
               . Jméno
funkce je zadáno skrz
                symbolový
                nebo
                êetëzcový

                token
               . Vîechny násle-
dující výrazy jsou vyhodnoceny a jsou pouùity jako argumenty volané funkce.

   Funkcí vrácená hodnota je pêiêazená zvláîtní proměnné RESULT. Není
chybou, kdyù není ùádná hodnota vrácena. V takovém pêípadë je proměnná
RESULT smazána, její inicializace je uçinëna zpëtnë skrz pokyn
                DROP
               .

   Spojení s funkcí je v okamùiku zavolání vytvoêeno dynamicky. Pêi hledání
volané funkce dodrùuje ARexx
                urçité poêadí
               .

 Pêíklad:

CALL CENTER name,delka+4,'+'  nebo  Call CENTER(name,delka+4,'+')

CENTER je volaná funkce. Výrazy jsou vyhodnoceny a jako argumenty vedeny
dále na
                CENTER
               .

CALL 'Rexxc:Pokus.rexx'  - volání jiného programu, za pêíkazem
                           CALL následuje êetëzec
CALL
                DELAY
                100     - volání funkce z pêipojené knihovny, za pêíkazem CALL
                      následuje Symbol

 !!!! Dûleùité upozornění: !!!!

   Pêíkaz CALL není nutné zadávat pêed kaùdé volání funkce, ale pokud takto
uçiníte, vyhnete se pozdëjîím problémûm. Pokud si myslíte,ùe je zbyteçné
tento pêíkaz psát pokaùde, musíte dodrùovat tyto pravidla:
```

1. Pokud pouùijete pêíkaz
   ADDRESS
   na zmënu
   host-adresy
   pouùíjte CALL
   radëji vùdy, protoùe pokud vámi zadaný arexxový port neexistuje, nebudou fungovat ani arexxové funkce. Kdyù pak zadáte ADDRESS Rexx, mûùete pêíkaz CALL potom jiù vypustit.
2. Je smazoêejmé, ùe pokud funkci nepouùíváte samostatnë, nesmíte CALL pouùít, napêíklad: SAY D2Ci(65), IF Pos('*',STR), a=x2c(b), ...
3. Pokud voláte
   interní funkci
   nebo
   externí funkci
   , pouùíjte CALL
   vùdy, co se mûùe stát si mûùete prohlédnout na dalîím pêíkladë:

```
/* Pokus s voláním procedûry */


                ADDRESS
                COMMAND
/* ... dalîí pêíkazy ... */
Tiskni('Ahoj')


                Exit
                0

Tiskni: Procedure

                Parse arg
                text

                Say
                'Procedûra tiskne:' text

                Return
                0
```

Program vypíîe toto:

```
Procedûra tiskne: Ahoj
0: Neznámý pêíkaz
0 failed returncode 10
  5 *-* Tiskni('Ahoj');
+++ Command returned 10
```

Jak sami vidíte, procedûra spuîtëna byla, ale na konci doîlo k velice zajímavé chybë... Dûvod je následující:

Procedûra vrátila nulu, ta je posléze poslána na
externí host
, ale ten
vrátí chybový kód, protoùe takový pêikaz nezná. Této chyby se lze zbavit i tak, ùe na konci funkce Tiskni() zmëníte "RETURN 0" na "RETURN ''".

Zajímavëjîí vîak bude, kdyù tentýù êádek zmënite na RETURN 'SAY' text. Pokud máte v adresáêi "C:" program "SAY", bude text nahlas pêeçten. Vrácený výraz je totiù opëï poslán na externí host, ten si sním tentoktrát ale poradí.

Snad vás tato çást neodradila od výuky ARexxu, protoùe zase aù tak sloùité to není. Arexxový interpret totiù pokud narazí na výraz, nejprve jej vyhodnotí,a pak postupnë podle
urçitého poêadí
zkouîí,zda je výsledek

interní funkcí
,
integrovanou funkcí
,atd. Takùe pokud vyhodnocený
výraz
nebude vyhodoncen jako existující pêikaz nebo funkce, dojde ↩ zákonitë
k chybë. Zatímco pokud pouùijete pêikaz CALL, není výsledek uù nijak vyhodnocován, ale je pouze uloùen do promënné RESULT.

1.40 Popis pêíkazu DO

```
                    | [<promënná>=<výraz>]|[<výraz>] [TO <výraz>] [BY <výraz>]
 DO | FOR <výraz>
    | [WHILE <
                 výraz
                >] | [UNTIL <
                 výraz
                >]
    | FOREVER
```

Pêíkazem DO je zavedena pêíkazová skupina, která bude vykonána jako blok. Oblast pêíkazu DO zahrnuje vîechny pokyny aù k jedinému moùnému pêíkazu END.

Kdyù po pêíkazu DO nenásleduje pod-klíçové slovo, je blok proveden jednou. Skrz zadání pod-klíçových slov mûùe být provádëní bloku opakováno aù do ukonçující podmínky. Iterativní (opakující se) DO pokyn se mnohdy oznaçuje jako smyçka (loop), protoùe se ARexx vrací zpët na místa, která uù proîel, aby vykonal pokyn jeîtë jednou. Pêíkaz DO se skládá z následujících sloùek:

* Inicializaçní výraz ve formë "<promënná>=<výraz>" definuje indexovou promënnou smyçky.
Výraz
je vyhodnocen, kdyù je oblast
pêíkazu DO poprvé aktivována. Výsledek je pêiêazen indexní promënné. Pêi následujících opakováních (iteracích) je výraz ve formátu "Promënná = Promënná + Inkrement" vyhodnocen, kde inkrement (zvýîení) je výsledek výrazu BY. Je-li zadán inicializaçní výraz, musí pêedcházet vîem pod-klíçovým slovûm.

* Po symbolu BY následující výraz definuje inkrement, který pêi

kaùdé následující iteraci bude pêidán k indexové promënné. Výraz musí dávat numerický výsledek, který je pozitivní nebo negativní a nemusí být celé çíslo. klíçové slovo BY nemusí být uvedeno, pak je pouùita standardní hodnota 1.

* Výsledek výrazu po TO urçuje vrchní (nebo spodní) hranici pro indexní promënnou. Pêi kaùdé iteraci je indexní promënná porovnávána s výsledkem od TO. Je-li inkrement positivní a promënná vëtîí neù hraniçní hodnota, pêeruîí se smyçka DO a pêedá êízení na pokyn následující po pêíkazu END. Smyçka je ukonçena také kdyù je inkrement negativní a indexní promënná leùí pod hraniçní hodnotou TO.
  Pêíklad: DO A=2 BY 2 TO 20 - A bude nabývat hodnoty 2,4,6..20

* Výraz po FOR musí pêi vyhodnocení dát pozitivní celé çíslo. Udává maximální poçet iterací. Smyçka bude ukonçena, okamùitë jak je dosaùeno hranice, nezávisle na hodnotë indexové promënné. Pêíklad DO A=2 BY 2 TO 20 FOR 5 - násobilka dvou bude pêedçasnë ukonçena po pátém prûbëhu smyçkou (A=10).

- Inicializovací výrazy BY, TO a FOR jsou vyhodnoceny jen pêi prvním aktivování pêíkazu; tím zûstane inkrement a hraniçní hodnoty nedotçeny bëhem provádëní. Hraniçní hodnota není bezpodmínëçnë nutná. Tak mûùe napêíklad s pêíkazem DO i=1 cyklus bëùet nekoneçnë dlouho.

* Klíçové slovo FOREVER mûùe být pouùito, kdyù je potêeba iterativní pêíkaz DO, ale není potêeba indexová promënná. Smyçka mûùe být opuîtëna pêíkazem
  LEAVE
  nebo
  BREAK
  obsaùeným ve smyçce.

* Výraz po WHILE je vyhodnocen na zaçátku kaùdé iterace a musí dávat
  booleovskou hodnotu
  . Iterace pokraçuje pêi výsledku 1.
  Jinak je smyçka ukonçena.

* Výraz po UNTIL je na konci kaùdé iterakce vyhodnocen a musí dávat booleovskou hodnotu. Pêíkaz pokraçuje s dalîí iterací, kdyù je výsledek 0, jinak je pêeruîen.
  (WHILE a UNTIL se navzájem vyluçují).

Mûùete vytvoêit têeba takový cyklus:

```
DO A=1 To 200 BY 4 WHILE P<8 FOR Pocet
...
END
```

Popis: Promënná "A" bude zvëtîována od 1 do 200 po çtyêech. Smyçka vîak bude ukonçena pêedçasnë za <Pocet> cyklû nebo pokud nebude splnëna podmína ùe P<8.

Program 12. Iterace.rexx Spusï program !

```
! Poznámka !
/* Pêíklady pro DO smyçku */
```

```
 LIMIT = 20; nummer = 1
 DO i = 1 to LIMIT for 10 WHILE nummer < 20
    nummer = i * nummer

                 SAY
                 "Iterace" i "nummer=" nummer

                 END
                 nummer = nummer/3.345; i = 0
 DO nummer for LIMIT/5
    i = i +1
    SAY "Iterace" i "nummer=" nummer
 END
```

Výstup je zde i s komentáêem, který se samozêejmë na obrazovce neobjeví.

```
 Iterace 1 Nummer = 1             /*  1 * 1 = 1  */
 Iterace 2 Nummer = 2             /*  2 * 1 = 2  */
 Iterace 3 Nummer = 6             /*  3 * 2 = 6  */
 Iterace 4 Nummer = 24            /*  4 * 6 = 24  */
 Iterace 1 Nummer = 7.17488789    /*  24/3.345 = 7.174888789  */
 Iterace 2 Nummer = 7.17488789    /*  Çíslo zûstane stejné  */
 Iterace 3 Nummer = 7.17488789    /*  Limit/5 = 20/5 = 4  */
 Iterace 4 Nummer = 7.17488789    /*  Operace se opakuje 4krát  */
```

Poznámka: Je-li pêítomna hraniçní hodnota pro FOR,i pêesto je pûvodní výraz vyhodnocen, nemusí ale dávat pozitivní, celé çíslo.

Podívejte se také na:

```
                 END

                 IF

                 SELECT

                 LEAVE

                 ITERATE

                 BREAK
```

1.41 Popis pêíkazu DROP

```
                 DROP <promënná> [<promënná> ...]
```

   Zadané promënné

```
                 symboly
                 jsou vráceny do stavu pêed jejich inicializací.
```

V tomto stavu odpovídá hodnota promënné samotnému jménu promënné. Není chybou, kdyù promënnou, která není inicializována, smaùete pêíkazem DROP. Je-li kmenový symbol smazán pêíkazem DROP, jsou také smazány hodnoty vîech od nëj odvozených sloùených symbolû.

Pêíklad:

```
 a = 123                  /* Pøiøadit - A - hodnotu */
 DROP a b                 /* Hodnotu z "a" a "B" smazat (DROP) */
 SAY a b    -> A B
```

Smazáním rozsáhlé kmenové promënné lze výraznë uîetøit pamët.

1.42 Popis pøíkazu ECHO

```
                ECHO [<výraz>]
```

Pøíkaz ECHO má shodný význam s pøíkazem SAY. Kopíruje výsledek výrazu, za který pøidá znak pro konec øádku, do
STDOUT
(napø. konzola)

Pøíklad: ECHO "Co neøíkáî !"

Bliùší informace u pøíkazu
SAY
!

1.43 Popis pøíkazu ELSE

```
                ELSE [;] [<alternativní pokyn>]
```

Pøíkaz ELSE vede alternativní podmínënou vëtev
IF
-ového pøíkazu. Je
platný jen v oblasti IF-ového pøíkazu a musí následovat po podmíněném pokynu THEN-ové vëtve. Kdyù nebude provedena THEN vëtev, bude zpracován pokyn následující po klausuli ELSE. (Jestliùe podmínka IF nebude splnëna.)

Pøíklad:

```
If i > 2 THEN SAY 'Skuteçnë?'
         ELSE SAY 'To jsem si myslel.'
```

Klausule ELSE se vztahuje vùdy bezprostøednë na pøedtím nacházející

IF
-ový pokyn. V pøípadë vkládání více podmínek do sebe mûùe ↩ vzniknout
problém nejednoznaçné interpretace pøíkazu ELSE, kdy není jasné, patøí-li do vnitøního çi vnëjšího cyklu. Tuto situaci reší pøíkaz
NOP
= ùádná
operace.

Pøíklad:

```
   IF A>10 Then IF A>5 Then Say 'Splñuje!'
                       Else NOP
           Else Say 'Nesplñuje'
```

Je naprosto jasné,co by se stalo, kdyby byl vynechán êádek s NOP.
Mûùete si to sami ovëêit na podobných pêíkladech...

Tento pêíkaz je souçástí pêíkazu
IF
, samostatnë nemá význam.

Podívejte se také na:
IF

SELECT

DO

1.44 Popis pêíkazu END

END [<jméno>]

Pêíkaz END uzavírá oblast pêíkazu
DO
nebo
SELECT
. Nepovinným parametrem <jméno> lze specifikovat, kterou do oblast tento pêíkaz ukonçuje. Argument musí souhlasit s názvem indexní promënné definované u pêíkazu DO, pokud symbol nesouhlaasí, dojde k chybë ç.
27
.

```
 DO i=1 to 5    /* Indexní promënná je i */
    SAY i
 END i        /* "i"-smyçku ukonçit */
```

Podívejte se také na
DO
a
SELECT
, bez nich nemá tento pêíkaz význam.

1.45 Popis pêíkazu EXIT

```
 EXIT [<výraz>]
```

Pêíkaz EXIT ukonçí provedení programu. Je platný kdekoliv v programu. Vyhodnocený výraz je vrácen zpët jako výsledek funkce nebo kommanda.

Zpracování EXIT-ového výsledku závisí na tom, zda volající úroveň (program, co spustil arexxový program) vyùaduje výsledný êetëzec a jestli volání pêiîlo z funkce nebo z kommanda:

* Byl-li vyùádán výsledný êetëzec, je potom výsledek výrazu zkopírován do reservovaného bloku v pamëti. Ukazatel na tento blok je potom sekundární výsledek volání.

* Nebyl-li êetëzec vyùádán a byl-li program zavolán jako kommando, je proveden pokus o jeho pêekonvertování na çíslo. Tato hodnota je potom vrácena jako primární výsledek. Sekundární výsledek je v tomto pêípadë 0. Tímto zpûsobem mohou být EXIT-ové informace od volající úrovnë interpretovány jako vrácená hodnota.

Poznámka:
Právë k druhému pêípadu dojde kdyù arexxový program voláte ze shellu. Pokud se nepodaêí pêevést výsledek na çíslo, vrátí dos tuto chybu:
Command returned 10/47: Arithmetic conversion error

Pêíklad:

```
EXIT                              /* ùádný výsledek není potêeba */
EXIT 'Chyba na êádku:' radek      /* vrátí êetëzec s informací.  */
EXIT 10                           /* Hláîení chyby */
```

1.46 Popis pêíkazu IF

```
IF <
výraz
> [;] THEN [;] [<podmíněný pokyn>] [; ELSE [;] <alternativní pokyn ↩
   >]
```

Pêíkaz IF je pouùíván ve spojení s pêíkazem THEN k provedení podmíněného pokynu a s pêíkazem

```
ELSE
```

k provedení alernativního pokynu.

Výsledek

```
výrazu
```

musí být booleovská hodnota. Je-li výsledek 1 (pravdivý), je proveden pokyn následující po THEN, jinak je proveden pokyn následující za ELSE, pokud je ovîem ELSE uvedeno. Za obëma klíçovými slovy mûùe následovat libovolný platný pokyn nebo blok

```
DO
/
END
```

. Lze také pouùít nulového pêíkazu

```
NOP
.
```

Klíçové slovo THEN ani

```
ELSE
```

nemusí následovat bezprostêednë po IF, ale

```
mûùou být zadány jako separátní klausule.  (Aù na dalîím êádku.)

Pêíklady:
 IF superA=cena            /* then nemusí následovat hned za podmínkou */
  Then Say 'OK'
  Else SAY 'NO'

 IF result < 0 Then exit   /* Vîechno hotovo */

Poznámka: "result" je návratová hodnota, kterou vrací nekteré externí fukce.
          Musíte vîak vracení aktivovat pêíkazem
                 OPTIONS
                 RESULTS
          Více  o  návratové  hodnotë  naleznete  u  pêíkazu
                 Call
                .
```

1.47 Popis pêíkazu INTERPRET

```
                 INTERPRET <výraz>
```

Pêíkaz INTERPRET jedná s výrazem, jako by to byl blok zdrojových pokynû. Výraz je vyhodnocen, a výsledek je proveden jako jeden nebo více programových pokynû.

Lze pouùít i pêíkazû jako je

```
                DO
                nebo
                SELECT
               , ale musí být uvnitê
```

interpetovaného výrazu kompletní. Je to jako kdyby jste programovali blok DO/END. Vykonávání interpretovaných pokynû lze také stejnë opustit pêíkazem

```
                BREAK
               .
```

Pêíkazy

```
                LEAVE
                 nebo
                ITERATE
                 lze pouùít jen v oblastech definovaných
```

uvnitê interpretovaného êetëzce.

Není sice chybou, kdyù jsou v INTERPRETováném êetëzci skokové znaçky, ale nemá to smysl, protoùe pêi pêenosu êízení je vyhledáváno jen mezi tëmi, co jsou definovány pêímo ve zdrojovém textu. Pêíkazem INTERPRET tedy nelze definovat novou

```
                interní funkci
               .
```

Pêíkaz INTERPRET mûùe být pouùit k dynamickému utváêení a provádëní programû. Programové fragmenty mohou být jako argumenty pêedány dále na funkci, která dále fragmenty INTERPRETuje.

```
Pêíklad:
 bef = 'SAY'            /* Pêíkaz */
 INTERPRET bef hallo    -> HALLO

 INTERPRET "DO p=1 FOR 5; Say 'A:'p; End"   /* ukázka vytvoêení smyçky */
```

1.48 Popis pêíkazu ITERATE

```
                ITERATE [<jméno>]

   Pêíkaz ITERATE ukonçuje aktuální iteraci (prûbëh smyçkou)
                DO
               -pêíkazu a
startuje  dalîí  iteraci.   To  znamená,  ùe  êízení programu je pêedáno aù
na pêíkaz
                END
               . Vîechny  pêíkazy mezi ITERATE a END jsou tedy pêeskoçeny.
Smyçka vîak není ukonçena.

   Pêíkaz  pûsobí  normálnë  na nejvnitênëjîí oblast DO.  Pokud vîak zadáte
nepovinný  parametr  <jméno>, lze opustit kteroukoliv smyçku.  <jméno> vîak
musí   souhlasit   s   názem   "indexní   promënné"   definované   u   pêísluînéhо
pêíkazu
                DO
               . Pokud zadáte neexistující promënnou, dojde k chybë ç.
                27
               .

   Kdyù se  pêíkaz ITERATE nenachází uvnitê iterativního DO pêíkazu (uvnitê
smyçky), dojde k chybë ç.
                22
               .

Pêíklad:
 DO i=1 to 5
    IF i = 3 THEN ITERATE i   /* i je název smyçky jeù se má opustit */
    SAY i
 END

   Podívejte se také na:
                LEAVE

                BREAK
```

1.49 Popis pêíkazu LEAVE

```
                LEAVE [<jméno>]

   Pouùití LEAVE vede  k okamùitému opuîtëní iterativní
                DO
```

oblasti, která
obsahuje tento pêíkaz. Êízení není pêedáno na pêíkaz
END
, jako u pêíkazu

ITERATE
, ale aù za nëj.

Pêíkaz pûsobí normálnë na nejvnitênëjîí oblast DO. Pokud vîak zadáte nepovinný parametr <jméno>, lze opustit kteroukoliv smyçku. <jméno> vîak musí souhlasit s názem "indexní promënné" definované u pêísluîného pêíkazu
DO
. Pokud zadáte neexistující promënnou, dojde k chybë ç.
27
.

Kdyù se pêíkaz LEAVE nenachází uvnitê iterativního DO pêíkazu (uvnitê smyçky), dojde k chybë ç.
22
.

Pêíklad:

```
 DO p=0
   DO i = 1 to 100
     IF i > 5 THEN LEAVE p     /* Maximální iterace pêi níù budou  */
   END                         /* obë smyçky opuîtëny              */
 END
```

Zpûsob uùití mûùete vidët také v ukázce
Násobilka
Podívejte se také na:
ITERATE

BREAK

1.50 Popis pêíkazu NOP

NOP

Jedná se o nulový (ùádný) pêíkaz, po jehoù pouùití se nic nestane. Slouùí vîak napêíklad k vázání ELSE klausulí na vnoêené
IF
pokyny.

Pêíklad:

```
                IF
                i = j THEN            /* První (vnëjîí) IF */
    IF j = k THEN a = 0   /* Vnitêní IF */
             ELSE NOP     /* Vázání na vnitêní IF */
ELSE a = a + 1            /* Vázání na vnëjîí IF */
```

Na pêíkladë je zcela zêejmé, ùe kdyby se nepouùilo êádku ELSE NOP,

nebylo by zêejmé, ke které podmínce poslední ELSE patêí...

Popis významu NOP naleznete také u pêíkazu
ELSE
.

1.51 Popis pêíkazu NUMERIC

```
                        | DIGITS [<výraz>]
        |   FUZZ [<výraz>]
 NUMERIC |       | [SCIENTIFIC]
        |  FORM | [ENGINEERING]
        |       | [[VALUE] <výraz>
```

Pêíkaz NUMERIC slouùí k nastavení voleb, které se vztahují na çíselnou pêesnost a formát. çíselné volby zûstanou pêi volání
interní funkce
zachovány.

* Výrazová volba DIGITS stanoví poçet platných míst pro pêesnost aritmetických výpoçtû. Výraz musí pêi vyhodnocení dát pozitivní, celé çíslo v rozsahu 1 - 14. Pokud vynecháte výraz, bude pêesnost nastavena na 9. Pozor, nejedná se o poçet desetinných míst, ale o celkový poçet míst bez desetinné teçky a nul na poçátku (pê. NUMERIC DIGITS 4 => 41.21)

* Výrazová volba FUZZ stanoví poçet platných míst, které mají být pêi çíselných porovnávacích operacích ignorovány. Zde se musí jednat o pozitivní, celé çíslo, které je menîí neù aktuální volba DIGITS.

* Volba FORM SCIENTIFIC stanoví k exponenciálnímu výpisu çísel vëdecký zpûsob zápisu. (napê. 8.854E-12)

* Volba FORM ENGINEERING volí k exponenciálnímu výpisu çísel technický zpûsob. Pêitom je çíslo normalizováno. tak, ùe jeho exponent je násobkem têí a mantissa v oblasti (1-999).

```
NUMERIC DIGITS 12         /* Pêesnë na 12 çíslic */
NUMERIC FORM SCIENTIFIC   -> 9.80676585E-46 /* Výsledek ve vëdeckém formátu */
NUMERIC FORM ENGINEERING  -> 980.676585E-48 /* Exponent je násobkem 3       */
```

Podívejte se také na funkce:
DIGITS()

FUZZ()

FORM()

PARSE NUMERIC

1.52 Popis pêíkazu OPTIONS

```
                          | FAILAT <výraz>
 OPTIONS | PROMPT <výraz>
         | RESULTS
         | CACHE
```

Tento pêíkaz slouùí k nastavení rûzných interních standardních hodnot.

Výraz po FAILAT nastaví hraniçní hodnotu, pêi jejímù dosaùení nebo pêekroçení se vrácené hodnoty kommand povaùují za selhání (failure), které lze odchytit
interuptem
FAILURE. Vyhodnocení výrazu musí dávat celé kladné çíslo. Implicitní nastavení pochází z volací úrovnë, pokud je tedy program volán ze shellu, je tato hodnota standartnë 10, ale mûùe být zmënëna jiù v shellu. Je tedy lepîí nastavit si svou hodnotu.

Výraz po PROMPT nastaví êetëzec,který bude pouùit pro pêíkaz
PULL
nebo

PARSE PULL
jako výzva k zadání vstupu.

Klíçové slovo RESULTS udává, ùe interpret má vyùadovat výsledný êetëzec, kdyù vysílá kommanda na
externí host
. Odpovëð se pak nachází v promënné "Result".

OPTIONS akceptuje také klíçové slovo CACHE, které slouùí k aktivování interní cache k meziukládání pokynû. Normálnë je cache aktivována. Programy tak bëùí rychleji.

Tímto pêíkazem êízené interní volby zûstanou zachovány i pêi volání funkcí. Pokud jej vîak pouùijete aù v
interní funkci
, neovlivní prostêedí z nëhoù byla funkce volána.

Pokud pouùijete pêíkaz OPTIONS bez argumentu, nastaví se vîechny volby na standartní hodnotu. Pêed volbami CACHE a RESULTS lze toké uvést klíçové slovo NO, které zapêíçiní zruîení daného nastavení. Mûùete tak napêíklad pêíkazem OPTIONS NO RESULTS doçasnë vypnout vracení výsledkû aniù ovlivníte nastavení FAILAT a PROMPT.

Pêíklad:

```
 OPTIONS FAILAT 10   /* chyba je povâùována za selhání aù pêi rc>=10 */
 OPTIONS PROMPT "Zadej pêíkaz, îéfe: "
 OPTIONS RESULTS     /* budou poùadovány výsledky */
 OPTIONS NO CACHE    /* zruîí vyrovnávací pamët */
```

1.53 Popis pêíkazu OTHERWISE

```
                OTHERWISE [;] [Podmíněný pokyn]

   Tento  pêíkaz  je  platný jen v oblasti pêíkazu
                SELECT
                a musí následovat
na konci vîech
                WHEN
               . Podmíněny  pokyn  bude  proveden jen v pêípadë, ùe
nebude splněna ùádná z pêedchozích podmínek zadaných u pêíkazû WHEN.

   Není  nutné,  OTHERWISE  vùdy  pouùít, ale musíte zajisti, ùe jej nebude
potêeba, protoùe jinak dojde k chybë ç.
                25
               .

Pêíklad:
 SELECT
    WHEN i=1 THEN SAY 'jedna'
    WHEN i=2 THEN SAY 'dvë'
    OTHERWISE SY 'jiné'
 END

   Tento pêíkaz je souçástí pêíkazu
                SELECT
               , samostatnë nemá význam.
```

1.54 Popis pêíkazu PARSE

```
                PARSE [UPPER] <Zdroj> [<
                îablona
               >]

 Jako zdroj lze zadat:

               |
                 ARG
                 |
               |
                 PULL
                 |
               |
                 EXTERNAL
                 |
               |
                 NUMERIC
                 |
 PARSE [UPPER] |
                 SOURCE
                 |
                 <îablona>
                              |
```

```
                VERSION
                |
              |
                VALUE <výraz> WITH
                |
              |
                VAR <promënná>
                |
```

Pêíkaz PARSE nabízí mechanismus k extrahování jedné nebo vícerých çástí êetëzcû z jednoho êetëzce a pêiêazení tëchto extraktû do promënných. êetëzec mûùe pocházet odkudkoliv napê. z argumentového êetëzce, výrazu nebo z konsoly (okna).

Popis tvorby îablon a vîeho, co s nimi souvisí najdete v 7. kapitole zabívající se

```
Syntaxní analýzou
a jejím pouùitím.
```

Pokud chcete napê. pouze pêidëlit tet naçtený z klávesnice nëjaké promënné, nemusíte se syntaxní analýzu zatím vûbec zabývat, staçí vám, kdyù napíîete

```
PARSE PULL
<promënná>.
```

Pokud pouùijete nepovinný parametr 'UPPER' bude zdrojový êetëzec pêed analýzou pêemënën na velká písmena. 'UPPER' musí být první token, který následuje po PARSE.

1.55 PARSE ARG - popis

```
PARSE
[UPPER] ARG <
îablona
>
```

Vstupní volba ARG vyvolá argumentové êetëzce, které byly zadány jako parametry pêi volání programu nebo interní funkce. Ty pak následnë rozkládá podle îablony a pêídëluje jednotlivé çásti promënným.

Rozdíl mezi samotným pêíkazem

```
ARG
```

a PARSE ARG je velice znatelný.
PARSE ARG totiù nechává malá písmena malými, zatím co samotné ARG je konvertuje na velká.

V originále manuálu (neù jsem jej upravil) bylo napsáno, ùe kommandové volání vykazuje bëùnë jen jeden jediný argumentový êetëzec,funkce naproti tomu aù 15. Zkouîel jsem proto toto tvrzení ovëêit. Sestavil jsem program, který byl volán s 26 parametry oddëlenými çárkou a tyto parametry byly uvnitê programu pouùity jako parametry pro volání interní funkce.
Pêedat parametry îlo, ale objevil jsem pár zajímavých vëcí:

```
1. Kommando nebo jiného programu lze opravdu volat jen z jedním argumentem,
   nelze zde pêi predávání  parametrû pouùít
                víceré îablony
               , lze  to vîak
   udëlat takto:  "
                ARG
                promA',''promB", coù  se vlastnë  bude  chovat jako
   kdyby jste  pêedávali více argumentû. Pokud  voláte  interní  funkci lze
   pêedání parametrû realizovat takto: "ARG promA,promB".

2. Voláte-li  z  programu jiný  program nebo pêímo program  ze Shellu, vadí
   ARexxu i tento  zápis:  ARG A','B','C.  Êeîení je snadné, musíte  pouùít
   název promënné o délce minimálnë dvou znakû.

Pêíklad:

  /* pêedávání parametrû programu pêi volání ze Shellu - TP */
  Parse Arg jmeno prijmeni','ulice','mesto
  Say 'Pan'prijmeni 'jehoù jméno je' jmeno 'bydlí ve mëstë' mesto', pêesnëjí:'  ↩
     ulice'.'

   Kdyù  tento  program  uloùíte  do  souboru, napêíklad: "RAM:ARG.rexx"  a
v Shellu napíîete: "RX RAM:ARG.rexx" a  doplníte  vaîe osobní udaje v tomto
poêadí: "<Jmémo> <Pêíjmení>,<Ulice>,<Mësto>",dozvíte  se od vaîeho poçítaçe
zajímavé vëci.
```

1.56 PARSE PULL - popis

```
                PARSE
                [UPPER] PULL <
                îablona
               >

   Vstupní volba PULL çte z
                STDIN
                (vstupní konzoly). Naçtený  êetëzec je
následnë  zpracován podle  zadané
                îablony
                a rozdëlen  promënným. Naçtený
êetëzec  není zmënën  na  velká  písmena, tak  jak  by jej  zmënil samotný
pêíkaz
                PULL
               . (pokud tedy nepouùijete klíçové slovo UPPER)

Pêi zadání
                vícerých îablon
                mûùe být najednou naçteno i nëkolik êádkû.

Pêíklad:
 /* Çtení ze vstupu - TP */
 Options Prompt 'Zadej nëjaky text s závorkou: '
 Parse Pull .'('text')'.
 Say 'V závorce se nachází:' text
```

Z êetëzce jenù zadáte bude vyñata pouze závorka a ta bude výpsána.

1.57 PARSE EXTERNAL - popis

```
                PARSE
                [UPPER] EXTERNAL <
                îablona
               >
```

Vstupní volba EXTERNAL çte z
STDERR
proudu,na který je smëêován výstup

chybových hláîení
a
monitorování
. Naçtený êetëzec (êádek) je pak zpracován podle zadané
îablony
.

Tato volba odpovídá volbë
PULL
, çte vîak z jiného
datového proudu
.

1.58 PARSE NUMERIC - popis

```
                PARSE
                [UPPER] NUMERIC <
                îablona
               >
```

Vstupní volba NUMERIC zdëlí aktuální numerické volby do êetëzce, a to v poêadí DIGITS, FUZZ, FORM, oddëleny od sebe mezerami.

Pêíklad:

```
 /*Numerický êetëzec je: "9 0 SCIENTIFIC"*/
 PARSE NUMERIC DIGITS FUZZ FORM .
 SAY digits    -> 9
 SAY fuzz      -> 0
 SAY form      -> SCIENTIFIC
```

1.59 PARSE SOURCE - popis

```
                PARSE
                [UPPER] SOURCE <
                îablona
               >
```

Vstupní volba SOURCE volá zdrojový êetëzec pro program. êetëzec je následovnë zformátován.

```
 {COMMAND | FUNCTION} {0|1} VOLAN ROZKLAD PREFIX HOST
```

Jednotlivé sloùky mají následující význam:

- {COMMAND | FUNCTION} udává, zdali program by lvolán jako kommando nebo funkce.
- {0|1} je booleovské znamení a udává, jestli byl výsledný êetëzec z volající úrovnë vyùádán. Viz.

```
                Popis logických operací
                - VOLAN je aktuální jméno arexxového programu. (Tak jak byl volán ↩
                   )
```

- ROZKLAD je celá cesta k volanému progamu
- PREFIX oznaçuje pêi hledacích operacích pouùívanou souborovou pêíponu (standardnë "REXX").
- HOST udává pûvodní host adresu pro kommanda. (napê. REXX)

Pêíklad:

```
 /* Informace - TP */
 Parse ARG zadano
 PARSE SOURCE Text

                Say
                Text

                Return
                Text':'zadano
```

Uloùte tento program têeba do "RAM:INFO.rexx" a vyzkouîejte program volat z nëkolika míst. Zkuste to têeba s Shellu pêíkazem: "RX RAM:INFO.rexx " nebo z jiného programu pêíkazovým êádkem: "Call 'Ram:INFO.rexx'.

Poznámka: Pêi spuîtëní programu z Shellu dojde k chybë zpûsobené tím, ùe program vrací návratovou hodnotu jako text a Shell ji nemûùe pêíjmout. Zatím co pokud budete tento program volat z jiného programu mûùete si návratovou hodnotu zjistit z promënné <RESULT>.

Podívejte se také na:

```
                CALL
```

1.60 PARSE VERSION - popis

```
PARSE
[UPPER] VERSION <
îablona
>
```

Vstupní volba VERSION vrací aktuální konfigurace arexxového interpretu je dána v následujícím formátu:

```
ARexx VERSION CPU MPU VIDEO FREQ
```

Jednotlivé komponenty mají následující význam:

* VERSION je çíslo verze interpretu ve formátu jako 1.15.

* CPU oznaçuje typ procesoru, který momentálnë provádí program. Moùné jsou napêíklad tyto hodnoty: 68000, 68010, 68020, 68030, 68040 atd.

* MPU je buð NONE, 68881 nebo 68882, závisle na tom, jestli je pêítomen matematický koprocesor.

* VIDEO je buð NTSC nebo PAL.

* FREQ udává síïovou frekvenci (60Hz nebo 50Hz).

Pêíklad:

```
/**/
PARSE VERSION data
Say Data             -> ARexx V1.15 68030 NONE PAL 50HZ
```

1.61 PARSE VALUE - popis

```
PARSE
[UPPER] VALUE [<výraz>] WITH <
îablona
>
```

Pouùijete-li pêíkaz PARSE tímto zpûsobem získáte mocný nástroj pomocí nëhoù mûùete rozkládat libovolné êetëzce. Výraz bude analyzován a nahrazen êetëzcem, jenù se dále úçastní anylýzy a rozkladu podle îablony. Klíçové slovo WITH je nutné, aby byl oddëlen výraz od îablony. Výsledek výrazu mûùe být skrz pouùití vícerých îablon vícekrát analyzován, výraz sám naproti tomu není znovu vyhodnocován. Pokud nehodláte pro rozklad pouùít výraz je jednoduîîí pouùít

```
PARSE VAR <promënná> <îablona>
```

.

Poznámka: Pokud výraz nezadáte bude vîem promënným pêidëlen prazdný êetëzec.

Pêíklad:

```
 /* */
 PARSE VALUE WITH S1 S2 S3
 SAY 'P:' S1 S2 S3 S4          -> P: S4
  /* promënným S1 S2 S3 byl pêidëlen prázdný êetëzec */

 Zpûsob pouùití mûùete vidët také v ukázce
                 Manipulace s êetëzci
                .
```

1.62 PARSE VAR - popis

```
                 PARSE
                 [UPPER] VAR <promënná> <
                 îablona
                >
```

Vstupní volba "VAR <promënná>" pouùívá hodnotu zadané promënné jako vstupní êetëzec jenù je analyzován a rozloùen podle zadané îablony. Hodnota zadané promënné se mûùe tímto procesem zmënit a to tehdy, jestliùeje její název patêí k pêiêazovacím

```
                 cílûm
                 v
                 îablonë
                .
```

Pokud vám nestaçí pouùití jen jedné promënné mûùete s výhodou pouùít komplikovanëjîí zápis

```
                 PARSE VALUE [<výraz>] WITH <îablona>
                .
```

Pêíklad:

```
 myvar = 1234567890
 PARSE VAR myvar 1 a 3 b + 2 c 1 d

                 SAY
                 a     -> 12
 SAY b     -> 34
 SAY c     -> 567890
 SAY d     -> 123456789
```

I takto komplikované îablony lze pouùít. Ti z vás, co programovali v jiných jazycích jistë tento pêíkaz velice ocení, protoùe kdyby neexistoval byl by rozklad trochu komplikovaný.

Zpûsob pouùití mûùete vidët také v ukázce

```
                 Manipulace s êetëzci
                .
```

1.63 Popis pêíkazu PROCEDURE

PROCEDURE [EXPOSE <promënná> [<promënná>] [...]]

Pêíkaz PROCEDURE je pouùit v interní funkci k vytvoêení nové symbolové tabulky. To zabrañuje, aby symboly (promënné), které jsou definovány ve spouîtëjícím prostêedí, byly bëhem provádëní funkce zmënëny. PROCEDURE je normálnë první pokyn ve funkci, povolený je ale také na kaùdém jiném místë. Nesmí se vîak vyskytnout dva pêíkazy v jedné funkci, protoùe dojde k chybë.

Pod-klíçové slovo EXPOSE nabízí selektivní jednání pro pêístup do symbolové tabulky spouîtëjící úrovnë a pro pêedání globálních proměnných na funkci. Promënné po klíçovém slovu odkazují na symboly v symbolové tabulce spouîtëjící úrovnë. Pozdëjîí zmëny tëchto proměnných jsou v prostêedí spouîtëjící úrovnë registrovány. Jednoduîe êeçeno, mûùete si sami zvolit, které promënné mohou být uvnitê procedûry (funkce) zmënëny a ty ostatní nemûùou být zmënëny, ani kdyù uvnitê procedûry pouùijete tytéù názvy.

V seznamu EXPOSE uvedené promënné mohou obsahovat kmenové symboly nebo sloùené symboly. V tomto pêípadë je vîak dûleùíte poêadí. Seznam EXPOSE Je totiù zpracován zleva doprava, sloùené symboly jsou rozîíêeny na základë hodnot, platících pro novou generaci. Vezmëme si napêíklad, ùe hodnota symbolu J byla v pêedchozí generaci 123 a J není v nové generaci inicializováno. Potom zpêístupní pokyn PROCEDURE EXPOSE J A.J symboly J a A.123, PROCEDURE EXPOSE A.J J naproti tomu A.J a J. Zadáte-li pouze A. bude zpístupnën celý kmen (pole proměných - viz
Promënné
).

Pêíklad:

```
 /* Procedûra - TP */
  Text = 'Amiga je super'

                Call
                Proc()

                SAY
                'Text:' Text ; Say ' STR:' STR

                Exit
                0

 Proc: PROCEDURE
  Text = 'Êetëzec zmënën.'
  STR= 'Promënná definována uvnitê programu.pokus'

                Return
                0
```

Na tomto jednoduchém prográmku mûùete experimentovat s procedurámi.
Zkuste napêíklad odstranit pêíkaz "PROCEDURE"...

Podívejte se také na:
CALL

1.64 Popis pêíkazu PULL

```
                PULL [<
                îablona
                >]
```

Pull je zkrácený pêíkaz PARSE UPPER PULL.

Çte z datového proudu
STDIN
, naçtený êetëzec pêevede na velká písmena
a analyzuje jej pomocí
îablony
. Pêi zadání
vícerých îablon
mûùe být
najednou naçteno i nëkolik êádkû.

Pêíkaz çte z konzoly, i kdyù není îablona k dispozici, ale naçtená data jsou ignorována.

Pêíklad:

```
 PULL Jmeno Prijmeni   /* çíst jméno */
```

Podívejte se také na:
PARSE PULL

1.65 Popis pêíkazu PUSH

```
                PUSH [<výraz>]
```

Tento pêíkaz pêipojí znak pro posun êádkû k výsledku výrazu a vloùí jej do zásobníku datového proudu
SDIN
. Je to jako kdyby jste do datového
proudu SDIN zadaly jeden êádek z klávesnice (napê. pêes okno shellu).

Zásobníkové êádky jsou êazeny podle pravidla "last-in, first-out" a mohou být çteny, jako kdyù jsou zadávány interaktivnë.

Po zadání pêíkazû

```
PUSH êádek 1
PUSH êádek 2
PUSH êádek 3
```

by datový proud byl çten v poêadí êádek 3, 2, 1.

PUSH umoùñuje nasazení
STDIN
datového proudu jako soukromý konceptní papír
k pêipravení dat pro dalîí zpracování. Napêíklad mohou být spojeny víceré soubory s pêipojením hraniçního znamení tím, ùe jsou jednoduîe pêeçteny

vstupní soubory, PUSHem postaveny do datového proudu a hraniçní znaky pêi potêebë pêipojeny.

Poznámka: Jednoduše êeçeno. Kaùdý výraz, který zadáte pêes PUSH bude uloùen a v okamùiku, kdy program poùádá o vstup (napê. pêíkazem
PULL
),
bude mu tento jiù vyhodnocený výraz pêedán jako êetëzec. Pêiçemù je vùdy upêednostñován ten poslední, ten nejçerstvëjîí.

Co se týçe priorit, je opakem tohoto pêíkazu pêíkaz
QUEUE
.

Pêíklad:

```
 /* push */
 DO i=1 to 5
    PUSH 'echo "êádek 'i'"'
 END

 Do Until Text=''
   PARSE PULL text

                 SAY
                 Text
 End
```

Zajímavé, mûùe být i toto pouùití:

```
/* PUSH */
 Say 'Ukládám do datového proudu...'
 PUSH 'run >nil: CLOCK'
 PUSh 'run >nil: SAY "Welcome to Amiga world"'
 Say  'Hotovo'
 Exit
```

Kdyù tento prográmek spustíte ze shellu,budou do
STDIN
(vstup) uloùeny
dva pêíkazy (Vçetnë zanku pro konec êádku - Enter.). Po îkonçení programu shell automaticky çeká na dalîí pêíkazy. Vy ale ùádný zadávat nemusíte, protoùe uù jsou dva uloùeny v zásobníku... Zkuste si to !

1.66 Popis pêíkazu QUEUE

```
QUEUE [<výraz>]
```

Tento pêíkaz pêipojí znak pro posun êádkû k výsledku výrazu a vloùí jej do zásobníku datového proudu
SDIN
. Je to jako kdyby jste do datového
proudu SDIN zadaly jeden êádek z klávesnice (napê. pêes okno shellu).

Zásobníkové êádky jsou êazeny podle pravidla "first-in, first out", jsou

tedy následnë çteny ve stejném poêadí v jakém byly pêíkazem QUEUE zadávány.

Zásobníkové êádky mohou být çteny, jako kdyù jsou zadávány interaktivnë.

V tomto pêíkladu by pêíkazy:

```
QUEUE êádek 1
QUEUE êádek 2
QUEUE êádek 3
```

byly çteny v poêadí 1, 2, 3.

Pêíkazem QUEUE uloùené êádky pêedcházejí interaktivnë zadaným êádkûm (tëm které zadáváte vy) a následují po êádcích uloùených pêíkazem PUSH.

Pêíklad:

```
 /* queue */
 DO i=1 to 5
    QUEUE 'echo "êádek 'i'"'
 END

 Do Until Text=''
   PARSE PULL text

                 SAY
                 Text
 End
```

Co se týçe priorit, je opakem tohoto pêíkazu pêíkaz
PUSH
.

1.67 Popis pêíkazu RETURN

```
                 RETURN [Výraz]
```

RETURN slouùí k opuîtëní
interní funkce
a pêedání êízení na místo
odkud byla funkce zavolána. Vyhodnocený výraz je pêedán jako výsledek funkce.

Není-li zadán výraz, mûùe v prostêedí volající úrovnë dojít k chybë. Zvláîï u funkcí, které jsou volány z
výrazu
, je nezbytné vrátit výsledný
êetëzec nebo çíslo. Není-li výsledek k dispozici, dojde k chybë ç.
16
.

Funkce, volané pêíkazem
CALL
, nepotêebují vracet výsledek.

Je-li tento pêíkaz zadán v základním prostêedí programu, není to chyba, RETURN zde odpovídá pêíkazu EXIT. U pêíkazu EXIT je také popsáno, jak je výsledný êetëzec pêeveden na externí volající úroveň.

Pêíklad:

```
 RETURN 6*7    /* Vrátí 42 */
```

Nyní nëco sloùitëjîího:

```
 /* Zjiîtëní nejvyîîího dëlitele çísla (menîího neù ono çíslo) - TP */
 Options prompt 'Zadej çíslo: '

                 PULL
                 cislo

                 SAY
                 Delitel(cislo)
 Say 'Z vracenou hodnotou lze dále poçítat.'
 Say '100-Delitel(cislo)*100 = '100-Delitel(cislo)*100

                 Exit
                 0

 Delitel:
                 PROCEDURE

                 ARG
                 cislo

                 Select

                 When
                 cislo=1 Then delit=1
     When cislo=0 Then
                 DO
                                        Say '?[1mNula jiù nemá dëlitele !?[0m'
                         cidlo=0; delit=1
                        End

                 Otherwise

                 DO
                 delit=2 TO cislo WHILE cislo//delit~=0 ; END

                 End
                   IF delit=cislo Then delit=1
 RETURN cislo/delit
```

Matematická podstata pêedchozího pêíkladu je napsána zvýrazněným písmem. Nebudu se spoléhat na vaîe matematické schopnosti a doufám, ùe nebudu ani pêeceñovat ty své, kdyù vám ji vysvëtlím:

Program neprovádí ùádný sloùitý rozklad, jak jsme se uçili jiù na základní îkole, ale vychází z toho, ùe poçítaç je ponëkud rychlejîí neù my. Jednoduîe dëlí zadané çíslo çíslem 2,3,4,5,6,... zadané çíslo do té

doby, neù narazí na dëlitele, kterým lze dëlit beze zbytku. Takovýmto zpûsobem nalezne nejmenîího dëlitele. A z nëj získat toho nejvëtîího je pêece maliçkost, staçí zadané çíslo tímto dëlitelem vydëlit...

Ostatní podmínky jsou vám jiù urçitë zêéjmé, jen oîetêují nedokonalost mého algoritmu pro prvoçísla a çísla 1,0... a to jeîtë nezkouîejte pouùít záporná çísla, protoùe vychází trochu nëpëkné výsledky... Ale nezoufejte, je to jen blbej program...

1.68 Popis pêíkazu SAY

```
SAY [Výraz]
```

Výsledek vyhodnoceného výrazu je (s pêipojeným znakem pro posun êádkû) vypsán na výstupní konzolu (datový proud

```
STDOUT
```

). Je-li výraz vynechán, je poslán prázdný êetëzec na konzolu.

Pêíkaz je ekvivalentní s pêíkazem

```
ECHO
.
```

Pêíklad:

```
SAY 'Odpovëð je' Hodnota
SAy '?[1mOdpovëð je' Hodnota'?[0m'  /* text bude vypsán tuçnë */
```

Jako souçást tisknutého textu lze zadat také speciální znaky, které se vîak nezobrazí, ale zpûsobí napêíklad zmënu vzhledu písma nebo posun kurzoru a dalîí uùiteçné vëci. Jedná se o Escape sekvence nebo ANSI znaky. Výpis vîech tëchto sekvencí a jeîtë nëco navíc, mûùete najít v dodatkovém souboru " ANSI ".

1.69 Popis pêíkazu SELECT

```
SELECT
```

SELECT zaçíná pêíkazový blok, který obsahuje jednu nebo víceré

```
WHEN
```

klausule a moùná i

```
OTHERWISE
```

klausuli. Po kaùdé klausuli následuje podmínëný pokyn. Jen jeden z pokynû uvnitê SELECT skupiny je proveden.

Nejprve jsou vyhodnocovány podmínky u jednotlivých WHEN, dokud není nalezena platná podmínka. Následující podmínky pak jiù nejsou kontrolovány! Není-li splnëná ùádná s podmínek, je proveden pokyn za OTHERWISE. Oblast SELECT musí být ukonçena pokynem

```
END
.
```

```
Pêíklad:
 SELECT

                 WHEN
                 i=1 THEN
                 SAY
                 'jedna'
     WHEN i=2 THEN SAY 'dva'

                 OTHERWISE
                 SAY 'jiné'

                 END
```

1.70 Popis pêíkazu SHELL

```
                 SHELL [ [<název portu> [<pêíkazový výraz>]] | [[VALUE] Výraz] ]

         | <jméno portu>  [<pêíkazový výraz>]
 ADDRESS | COMMAND [<pêíkazový výraz>]
         | [VALUE] <výraz>
         | (bez argumentu)

Pêíkaz SHELL má stejný význam i syntaxi jako
                 ADDRESS
                .

Pêíklad:
 SHELL edit    /*Nastavit host na 'EDIT'*/
```

1.71 Popis pêíkazu SIGNAL

```
                 SIGNAL {ON | OFF} <interrupt>
 SIGNAL [VALUE] <název návëstí>
```

SIGNAL {ON | OFF} <interrupt> êídí status interních poznávácích znaçek pêeruîení (interrupt flags). Pêeruîení umoùñují programu, pêi výskytu urçité chyby, chybu rozpoznat a udrùet si êízení. V tomto pêípadë musí po SIGNALu následovat klíçové slovo ON nebo OFF a jedno z následujících podmínkových klíçových slov. V podmínkovém symbolu zadaná pêeruîovací znaçka je potom zmënëna na zadaný status.

Nyní následuje struçný pêehled vîech interrpuptû, podrobnëjîím popisem se zabývá kapitola

```
                 Pêeruîení
                .

BREAK_C    Zachycení pêeruîení CTRL-C.
BREAK_D    Zachycení pêeruîení CTRL-D.
BREAK_E    Zachycení pêeruîení CTRL-E.
```

```
BREAK_F    Zachycení pêeruîení CTRL-F.
ERROR      Host-kommando vrátilo chybovou hodnotu rûznou od nuly.
FAILURE    Host-kommando vrátilo chybovou hodnotu, která je vëtîí neù mez
           nastavená pêíkazem
                OPTIONS
                FAILAT.
HALT       Externí poùadavek HALT byl poznán.
IOERR      Vstupní/výstupní systém zjistil chybu.
NOVALUE    Byla pouùitá neinicializovaná promënná.
SYNTAX     Syntaxní nebo bëhová chyba.
```

Podmínková klíçová slova jsou interpretována jako skokové znaçky, na které je pêeneseno êízení, kdyù nastoupí vybraná podmínka. Je-li napêíklad aktivováno ERROR pêeruîení a kommando vrátí hodnotu rûznou od nuly, pêenese ARexx êízení na skokovou znaçku ERROR: . Tato znaçka musí být pêirozenë v programu definována. Jinak dojde okamùitë k syntaxní chybë, a program bude opuîtën.

```
Upozornëní: Pokud v programu pêidáváte knihovnu do seznamu knihoven, musíte
            pêíkaz SIGNAL ON <interrupt> dát aù za funkcí
                 ADDLIB()
                . Funkce
            totiù generuje návratovou hodnotu která mûùe zpûsobit pêedání
            êízení na procedûru pro odchycení chyb. Pêitom k ùádné chybë
            nedoîlo. Uîetêíte si tak spoustu práce a çasu...
```

V SIGNAL [VALUE] <název návëstí> jsou po SIGNALu následující tokeny vyhodnoceny jako výraz. Je generováno okamùité pêeruîení, které pêedá êízení na výsledkem výrazu definovanou skokovou znaçku. Pêíkaz má efekt "poçítaného GOTO".

```
Jakmile nastoupí pêeruîení, jsou vîechny toho çasu aktivní êídící oblasti
(
                 IF
                ,
                 DO
                ,
                 SELECT
                ,
                 INTERPRET
                 nebo interaktivní
                 TRACE
                ) pêed pêedáním
```

êízení inaktivovány. Pêedání nemûùe být tedy pouùito ke skoku do DO smyçky nebo jiné êídící struktury. Jelikoù SIGNAL pûsobí jen na êídící struktury aktuálního prostêedí, je zcela bezpeçné jeho pouùití uvnitê

```
                 interní funkce
                . Status volajícího prostêedí zûstane nedotçen.
```

Zvláîtní promënná SIGL je, jakmile dojde k pêenosu êízení, nastavena na aktuální çíslo êádku. Program mûùe na základë pêezkouîení promënné SIGL zjistit, který êádek byl provádën pêed pêenosem êízení.

Zpûsobí-li ERRORová nebo SYNTAXní podmínka pêeruîení, je zvláîtní promënná RC nastavena na chybový kód, co pêeruîení zpûsobil. V pêípadë ERRORové podmínky udává tento kód stupeñ závoùnosti chyby a SYNTAXní podmínka udává arexxový chybový kód. Bliùîí vysvëtlení se dozvíte v çásti

```
zabívající se
                chybami
               .

 Pêíklad:

SIGNAL on error     /*Aktivovat pêeruîení*/
SIGNAL off syntax   /*SYNTAX inaktivovat*/
SIGNAL start        /*Skoçit na start*/

Bliùîí popis pêeruîení najdete v
                6.kapitole - Pêeruîení
               .

V této souvislosti by se vám mohly hodit tyto funkce:
                ERRORTEXT()

                SOURCELINE()
```

1.72 Popis pêíkazu TRACE

```
                       | [ {?|!} ] [<volba>]
 TRACE | VALUE <výraz>
       | <çíslo>
```

Tento pêíkaz slouùí k zapnutí nebo vypnutí monitorování.

Zadáním parametru <volba> se zapne reùim nìkterý reùim monitorování. Zadáte-li pêed volbu jeden z moùnách znakû, aktivujete interaktivní reùim (?) nebo komandová záchytka (!).

Pêíkaz TRACE VALUE <výraz> umoùnuje nastavení monitorování podle výhodnocení výrazu. Mûùete tak napêíklad umoùnit uùivateli, aby si reùim sám vybral. Pêíklad:

```
PARSE PULL
Mod ; TRACE VALUE Mod.
```

Poslední moùností je zadání (TRACE <çíslo>). Çíslo musí být vùdy celé, ale mûùe být kladné i záporné:

- Pokud zadáte <çíslo> kladné, nebude se interaktivní monitorování dané mnoùství klauzulí pozastavovat. Poçítány jsou jen ty klauzule, které provádë nìjakou zmìnu nebo akci. Napê. funkce

```
Delay()
```

nebude poçítána.

- Pokud zadáte <çíslo> záporné, bude interarktivní monitorování, pro daný poçet êádkû od pêíkazu TRACE, deaktivováno.

1.73 Popis pêíkazu WHEN

```
                WHEN <
                výraz
               > [;] THEN [;] [<podmíněný pokyn>]

   Pêíkaz WHEN je srovnatelný s
                IF
               , je vîak platný jen v oblasti
                SELECT
               .

   Vîechny
                výrazy
                u jednotlivých pokynû WHEN jsou postupnë vyhodnocovány
jako "podmínky"  a musí tudíù dávat
                booleovský výsledek
               . Je-li výsledek 1
(pravda),  je podmíněný pokyn proveden, a êízení jde na ENDový pokyn, který
ukončí  SELECT.  Je  dûleùité  uvëdomit si, ùe pokud se v oblasti SELECT -
WHEN  nachází  více  podmínek, které vrací 1 (jsou splněny), bude provedena
pouze jedna a a to ta první a ostatní budou vynechány !!!

   THEN  nepotêebuje (pêesnë jako u IF) být souçástí té samé klausule, tzn.
ùe mûùe být aù na dalîím êádku.

Pêíklad:

                SELECT
                  WHEN (i>100 & i<1000 ) | P=0
    THEN Say 'Promënná "i" je z intervalu (100,1000) nebo se P rovná 0'
   WHEN i<j THEN
                SAY
                'Menîí'
   WHEN i=j THEN SAY 'Rovno'

                OTHERWISE
                SAY 'Vëtîí'

                END
                  Tento pêíkaz je souçástí pêíkazu
                SELECT
               , samostatnë nemá význam.
```

1.74 5. kapitola - Funkce

```
               F U N K C E  (Procedûry)
```

Pod funkcí se rozumí program nebo skupina pokynû, který/které jsou provedeny, kdyù je jméno funkce v urçitém kontextu zavoláno. Funkce mûùe být çást interního programu, knihovny nebo separátního externího programu. Funkce jsou dûleùitou souçástí pêi modulárním programování, protoùe umoòñují výstavbu komplexních programû z menîích, jednoduîîích modulû

(çástí), které se lépe a rychleji opravují a rozvíjejí.

Poznámka: Funkce se od pêíkazu liîí pêedevîím tím,ùe vrací nëjaký výsledek, ale také tím, ùe si mûùete sami tvoêit nové funkce.
Procedûry je obdobný název pro funkci,çasto se sním mûùete setkat v odborné literatuêe. Nevím, jestli je mezi tëmito pojmy nëjaký rozdíl. Já pouùívám výraz "Funkce" jen tehdy, jestliùe se jedná o program (çast programu), který je volán jen z toho dûvodu, aby vrátil nëjakou hodnotu (napê. provedl vypoçet). Pokud ale slouùí k tomu, aby napêíklad vypisoval text na obrazovku, nazývám ho jiù "procedûra".

Nejprve je dûleùité êíci si nëco o tom, jak se taková funkce dá volat.

Volání funkce
Vîechny funkce se dají rozdëlit do 3 základních skupin.

Interní funkce
- definované v arexxovém programu.

Integrované funkce
- pochází z ARexxu jako programovacího jazyka.

Externí knihovny funkcí
- speciální knihovny s arexxovými funkcemi.

Vîechny knihovny, jejichù funkce lze momentálnë pouùívat, jsou zapsány v tzv. "
Seznam knihoven
", který je spravován pêimo interpretem ARexxu.

V této souvislosti se zmíním také o tzv. "
Clipboardu
", který je také zpravován externím prostêedím. A je jednou z moùností jak pêenáîet mezi programy infomrace...

Poçet funkcí, ale i pêíkazû lze také rozîíêit pomocí
Externích hostû
.

Pokud zavoláte nëkterou funkci, musí interpet nejprve zjistit zda je tato funkce funkcí ARexxu nebo je souçásti programu, knihovny a nebo ji dokáùe zpracovat externí host.
Pêi tomto vyhledávání je dodrùováno
urçité poêadí
.

Neù se podíváme na popis samotných funkcí, je nutné seznámit se s základy
Syntaxe
.

Nyní se se vîemi funkcemi seznámíme podrobnëji:

Abecední seznam integrovaných arexxových funkcí
Abecední seznam funkcí v knihovnë

Poznámka: Pokud budete chtít funkce procházet jednu po druhé, mûùete pouùít listování. (Poloùka v horní liîte gadgetû. (angl. Browse))

1.75 Popis Volání funkce

5.1 Volání funkce

Bëhem arexxového programu je funkce definována jako symbol nebo êetëzec, po kterém bezprostêednë následuje otevírající kulatá závorka. Symbol nebo êetëzec (interpretován jako literál) udává jméno funkce.Po otevêené závorce následuje seznam argumentû. Pokud je argumentû více, oddëlují se çárkami. Pak následuje uzavírající kulatá závorka.Pokud se funkci argumenty pêedávat nemusí, lze je vynechat.

Platná volání funkcí jsou:

```
CENTER('Nadpis',20)
ADDRESS()
'ALLOCMEM'(256*4,1)
DELAY 100  nebo DELAY(100)
```

Vîechny argumentové výrazy jsou vyhodnocovány postupnë, z toho výsledné êetëzce jsou pêedány dále funkci jako seznam argumentû. Kaùdý argumentový výraz (çasto jen jediná literální hodnota) mûùe obsahovat aritmetické nebo êetëzcové operace nebo jiná volání funkcí. Argumentové výrazy jsou vyhodnoceny v poêadí zleva doprava. Funkce mohou být také volány pêíkazem

CALL

. Pêíkaz CALL (viz.4.2) mûùe být pouùit k volání funkce, která ↩ moùná nevrátí hodnotu.

Poznámka: Menîí problém s voláním funkcí je popsán u popisu pêíkazu

CALL

, který takovým problémûm zabrañuje.

1.76 Druhy funkcí -> Interní funkce

5.2.1 Interní funkce

Ty to funkce si uùivatel mûùe definovat sám. Oznaçují se skokovou znaçkou v programu,tzv. návëstím. To mûùe být libovolný název, jenù musí být tvoêen zemito znaky: "A-Z,a-z,$,_,!,@,#". Pak následuje dvojteçka. (Napê. TISKNI:)

Pêi vyvolání interní funkce vytvoêí ARexx nové prostêedí, aby prostêedí pûvodní úrovnë zûstalo zachováno.Nové prostêedí pêebírá hodnoty pêedchûdce, pozdëjîí zmëny nemají ale vliv na pêedchozí prostêedí.

Jsou chránëny tyto hodnoty:

* Aktuální a pêedchozí
  host
  adresy

* Pêednastavení pro
  NUMERIC
  DIGITS, FUZZ a FORM

* Volba
  monitorování
  , trasovací znaçka a interaktivní znaçka

* Vîechyny
  interrupty
  aktivované pêíkazem
  SIGNAL
  * Vîechny volby nastvené pêíkazem
  OPTIONS
  PROMPT

* Interní poçítadlo êízené funkcemi
  TIME
  ('E') a TIME('R')

Nové prostêedí vîak neobsahuje automaticky novou symbolovou tabulku, tudíù volané funkci stojí k dispozici vîechny promënné z pêedchozího prostêedí. Pêíkaz

  PROCEDURE

  mûùe být pouùit k vytvoêení vlastní symbolové tabulky,aby byly hodnoty symbolû z volající úrovnë uchránëny. PROCEDURE mûùe být pouùit k vyuùití jednoho jména promënné ve dvou oblastech s rûznými hodnotami. V provedení interní funkce je pokraçováno, dokud není nalezen pêíkaz

  RETURN

. Na tomto místë je nové prostêedí inaktivováno,a êízení se vrací zpët k tomu bodu, odkud byla funkce zavolána . Výraz zadaný s pêíkazem RETURN je vyhodnocen a vrácen zpët jako výsledek funkce.

1.77 Druhy funkcí -> Integrované funkce

5.2.2 Integrované funkce

Jako souçást systému nabízí ARexx rozsáhlou knihovnu pêeddefinovaných funkcí. Tyto funkce jsou stále k dispozici a pêizpûsobeny k spolupráci s vnitêními datovými strukturami. Vîeobecnë bëùí integrované funkce znaçnë rychleji neù odpovídající interpretované funkce. Z tëchto dûvodû je jejich pouùití doporuçeno. (Bohuùel vám jeich moùnosti brzy pêestanou staçit.)

Nëkteré integrované funkce vytváêí a spravují externí AmigaDOSové soubory. U souborû je bráno v potaz jejich logické jméno. Pêi tomto jménu, které je souboru pêi otevêení pêiêazeno, je psaní velkých a malých písmen dûleùité. Pûvodní vstupní a výstupní datové proudy dostanou jména

STDIN
(standardní
vstup) a
STDOUT
(standardní výstup). Teoreticky mûùe být libovolnë mnoho souborû souçasnë otevêených, ovîem pracovní pamët není nekoneçná, takùe nëkdy na tyto hranice narazíte, kdyù se pokusíte otevêít nekoneçnë mnoho souborû najednou. Pêi opouîtëní programu se vîechny soubory automaticky zavêou.

Viz. funkce
OPEN()

CLOSE()

1.78 Druhy funkcí -> Externí knihovny funkcí

5.2.3 Externí knihovny funkcí

Knihovna funkcí (shared library) je sbírka vícerých funkcí, která je stavëna stejnë jako ostatní amigácké knihovny.

Knihovna musí být zapsána v LIBS: , je jedno jestli na harddisku nebo v pamëti. Z harddisku jsou podle potêeby nahrány a otevêeny.

Knihovna musí být pro pouùití ARexxem specielnë upravena. Kaùdá knihovna funkcí musí mít jméno knihovny, hledací prioritu, relativní vstupní pozici a çíslo verze. Kdyù ARexx hledá funkci, otevêe interpret kaùdou knihovnu a pêezkouîí "query" vstupní bod. Tento vstupní bod musí být celoçíselnou relativní pozicí vzhledem k bázi knihovny. Vrácená hodnota zkuîebního volání udává, jestli vyùadovaná funkce byla nalezena. Jestli ano, tak je zavolána interpretem s pêedáním parametrû, a výsledek funkce je vrácen. Není-li nalezena, je vrácen chybový kód pro "funkce nenalezena", a hledání pokraçuje v dalîích knihovnách na seznamu. Knihovny funkcí jsou po pouùití zavêeny, aby systém disponoval celou pamëtí.

Funkce pro práci z knihovnami:
ADDLIB()

REMLIB()
O pêipravenosti knihovny se lze informovat funkcí
SHOW()
nebo rozîíêenëjîí
funkcí knihovy RexxSupport.library,
SHOWLIST()
.

1.79 Seznam knihoven

5.2.3.1 Seznam knihoven

Residentní arexxové procesy spravují seznam momentálnë pouùitelných knihoven funkcí a host funkcí - tzv. seznam knihoven. Aplikaçní programy mohou jakoukoliv knihovny funkcí pêipojit nebo odstranit.

Seznam knihoven je spravován jako struktura podle hodnot priorit. Kaùdý zápis vlastní jemu pêiêazenou hodnotu (priorita pêi hledání) 100 aù -100 (nejvëtîí aù nejmenîí). Zápisy mohou být pêipojeny s vhodnou prioritou, aby byl êízen rozklad funkçních jmen (rozklad-analýza,práce se jménem funkce). Knihovny s vyîîími hodnotami priorit jsou prohledány jako první. Uvnitê zadané úrovnë priorit jsou prohledávány nejprve knihovny, co byly nejprve pêipojeny. Úrovnë priorit mají svûj význam, kdyù víceré knihovny vykazují identické definice funkçních jmen, protoùe v poêadí dole stojící funkce by nebyly jinak zavolány.

```
Knihovnu lze pêipojit funkcí
                 ADDLIB()
                 nebo zruîit funkcí
                 REMLIB()
                .

O pêipravenosti knihovny se lze informovat funkcí
                 SHOW()
                 nebo rozîíêenëjîí
funkcí knihovy RexxSupport.library,
                 SHOWLIST()
                .
```

1.80 Popis externích funkçních hostû

5.2.4 Externí funkçní hosty

Dalîí moùností,jak získat nové funkce je "Externí host". Pokud máte program který disponuje arexxovým portem, znáte jeho název a funkce nebo pêíkazy které lze pouùít, tak máte vyhráno. Staçí se k portu jen pêipojit pêíkazem

```
                 ADDRESS
                . O existeni portu se dá pêesvëdçit funkcí
                 SHOW()
                  nebo

                 SHOWLIST()
                , která je souçástí knihovny "RexxSupport.library"
```

Funkçnímu hostu pêiêazené jméno je jméno jeho veêejnë pêístupného message-portu. Volání funkcí jsou pêedávány ve formë zprávového paketu na host. Potom zjiîîuje onen host, zdali pozná jméno funkce. Rozklad jména je vnitêní host-proces. Funkçní hosty nabízí pêirozený gateway postup k implementaci dalîích volání funkcí na ostatní stroje v síti. Residentní arexxový proces je funkçní host a je instalován v seznamu knihoven s prioritou -60.

1.81 Poêadí pêi hledání

5.3 Poêadí pêi hledání

Následující çást popisuje zpûsob hledání funkcí, které závoláte.

Funkçní svazky jsou v ARexxu vytvoêeny v okamùiku zavolání funkce. Urçité poêadí pêi hledání je dodrùováno, dokud není nalezena funkce, která souhlasí se jmenným symbolem nebo êetëzcem. Není-li funkce nalezena, je výsledkem chyba, a vyhodnocení výrazu je ukonçeno.

Upozornëní: Tiskni() -> jmenný symbol -> ARexx hledá funkci s názvem: TISKNI
'Tiskni'() -> êetëzec -> ARexx hledá funkcí s názvem: Tiskni
Poznámka: Doporuçuji funkce volat jako jmenný symbol, napê. Call AddLib(...

Plné poêadí pêi hledání funkce vypadá následovnë:

Interní funkce — Zdrojový text programu je prohledán, jestli neobsahuje skokovou znaçku, která odpovídá jménu funkce. Je-li nalezena, je vytvoêeno nové prostêedí a êízení pêechází na onu skokovou znaçku.

Integrované funkce — Knihovna s integrovanými funkcemi je prohledána, jestli neobsahuje zadané jméno. Vîechny tyto funkce jsou definovány jmény s velkými písmeny.

Knihovny funkcí a Funkçní hosty — Pouùitelné knihovny funkcí a funkçní hosty jsou spravovány v seznamu knihoven, který je prohledáván podle poêadí priorit, dokud není funkce nalezena nebo dosaùeno konce seznamu. (viz. Seznam knihoven
Funkçní hosty jsou volány protokolem pro pêenos zpráv, který se dá srovnat s protokolem, pouùívaným pro kommanda. Mohou být pouùity jako gateway pro dalîí volání procedur na jiných poçítaçích uvnitê sítë. (viz. Extení hosty
)

Externí arexxové programy — Poslední krok v poêadí je hledání externího arexxového programového souboru. Za tímto úçelem je poslána zpráva na residentní arexxový proces. Hledání zaçíná vùdy v aktuálním adresáêi a poté ta samá cesta,jako pêi pêedchozím volání arexxového programu. Pêi porovnávání jmen nehrají velká çi malá písmena ùádnou roli, také koncovku "rexx" nemusíte pêi volání uvádët. Pokud voláte funkcí, která ve svém jménu obsahuje nedovolené znaky, musíte funkcí volat jako êetëzec. Napê:

```
Call Najdi()   /* závorka není nutná      */
Call 'Poçítej'(1+2) /* závorka není nutná */
Call 'Ram:Opoçet' 10,'Zbohem'
```

Prosíme o vënování pozornosti faktu, ùe pêi proceduêe porovnávání jmen funkcí pêi nëkterých krocích je rozliîováno mezi velkými a malými písmeny a pêi nëkterých ne. Jestli to je pêípad porovnávací procedury v knihovnë

funkcí nebo u funkçního hostu, to je v moci programátora. Funkce, v jejichù názvu jsou malá a velká písmena, musí být volány êetëzcovým tokenem, protoùe jako symbol by byl název pêemënën na velká písmena.

Uplné poêadí pêi hledání je dodrùeno jen tehdy, jestli je jméno funkce definováno symbolovým tokenem. Hledání interních funkcí je pêeskoçeno, jestliùe jméno stojí v êetëzcovém tokenu. To dává externím nebo interním funkcím moùnost, obsadit jména integrovaných funkcí,jak ukazuje následující pêíklad:

```
CENTER:         /*Interní "CENTER"*/
ARG êetëzec,Délka     /*Argumenty*/
Lang = MIN(Lang,60)     /*Zmënit délku*/
RETURN 'CENTER'(êetëzec,Délka)
```

Zde byla integrovaná funkce CENTER() po zmënění délkového argumentu nahrazena interní funkcí.

1.82 Popis Clipboardu

5.4 Arexxový Clipboard

Clipový seznam je vîeobecnë pêístupný, je to nëco jako mezipamëï (clipboard), pouùitelný pro komunikaci rûzných programû. Mnoho funkcí pouùívá tuto mezipamëï k volání rûzných druhû dat, napê. pêeddefinované konstanty nebo êetëzce.

Clipboard spravuje êadu párových pojmû (<jméno>,<hodnota>), které jsou mnohostrannë pouùitelné. Kaùdý záznam mûùe být lokalizován podle svého jména. Jména by mëla být volena tak, aby nemohlo dojít ke konfliktûm s ostatními programy, které zrovna pouùívají tentýù název. Poçet zápisû do clipboardu je libovolný.

Moùné pouùití clipboardu je napêíklad naçítání pêeddefinovaných konstant do arexxového programu.

```
  Pêíklad:   Pokud pêíkazem
                 SETCLIP()
                  uloùíte  do  clipboardu pod jménem
             'konstanty' následující êetëzec:

   Pi=3.1415926; e=2.718; sqrt2=1.414  . . .
```

Takovýto êetëzec mûùe být na základë svého jména ('konstanty') vyvolán pêes integrovanou funkci

```
                GETCLIP()
```

a uvnitê programu INTERPRETován.
Pêiêazující pokyny uvnitê êetëzce by pak uçinily potêebné definice konstant. Pêíklad:

```
/* INTERPRETace poloùky s clipboardu */
 konstanty = GETCLIP('konstanty')

                 INTERPRET
```

```
                konstanty      /* Êetëzec bude vykonán jako arexxový pêíkaz. */
```

Êetëzec by samozêejmë nebyl omezen jen na pêiêazovací pokyny, ale mohl by obsahovat i libovolné arexxové pokyny. Clipboard by mohl také zásobovat êadu programû inicializacemi a jinými úlohami.

Residentní proces podporuje operace pêipojování a mazání k spravování clipboardu. Co se týçe jmen v párových pojmech (<jméno>, <hodnota>), je pêedpokládáno, ùe obsahují velká a malá písmena a v seznamu se jiù vícekrát nevyskytují. Pokus, pêipojit êetëzec s jiù pouùitým názvem , vede jen k aktualizaci (pêepsání) pûvodního êetëzce.

Dûleùíte je, ùe clipboard není spravován programem, který do nëj provádí zápis, v clipboardu tudíù zûstanou data i po ukonçení programu. Smazat jej lze funkcí

```
                SETCLIP()
                s uvedením prázdného êetëzce.
```

Seznam vîech poloùek v clipboardu lze získat funkcí

```
                SHOW()
               .
```

!!! Upozornëní !!!

Workbench má také clipboard v nëmù lze uchovávat napêíklad text nebo obrázky. Tento clipboard vîak není totoùný s tím, který má ARexx. Pokud chcete manipulovat s clipboardem Workbenche, mûùete pouùít napêíklad funkce obsaùené v externí knihovnë "RexxTricks.library"

1.83 Popis syntaxe

5.5.1 Syntaxe

Syntaxe je vlastnë popis pouùití funkce, udává nám, co máme pêi volání funkce zadat, abychom obdrùely oçekávaný výsledek. V následující çásti je posáno nëkolik základních pravidel zápisu syntaxe.

Volitelné argumenty stojí v hranatých závorkách a mají normálnë standardní hodnotu, která je pouùita, kdyù není argument zadán explicitnë. Je-li zadáno klíçové slovo volby jako argument, má význam jen první znak. Pêi klíçových slovech volby není rozliîováno mezi velkými a malými písmeny.

Mnohé funkce umoùñují vyplñující znakové argumenty. Vyplñující znaky jsou vloùeny, aby vyplnily nebo nahradily mezery. Pêi takovém vyplñujícím znakovém argumentu je jen první písmeno argumentového êetëzce dûleùité. Pêi zadání prázdného êetëzce je pouùit standardní vyplñovací znak (obvykle mezera).

V nëkteých pêíkladech se vyskytne tato îipka "->", veskuteçnosti není souçátí funkce, ale oznamuje nám jakou hodnotu funkce vrátí.

Pêíklad:

```
SAY ABS(-5.35) -> 5.35

Toto znamená, ùe SAY ABS(-5.35) pêi vyhodnocení dá 5.35 .
```

1.84 Seznam funkcí

```
                Funkce - popis
               5.5.2 Abecední rejstêík Integrovaných funkcí a funkcí knihovny  ↩
                  RexxSupport
-------------------------------------------------------------------------------
  Integrované funkce

        Abecední seznam               Syntaxe
 A
 ---------------------------

                ABBREV()
                - (<dlouhý êetëzec>,<krátky êetëzec>[,<délka>])

                ABS()
                - (<çíslo>)

                ADDLIB()
                - (<jméno>,<priorita>[,<offset>,<verze>])

                ADDRESS()
                - ()

                ARG()
                - ([<çíslo>][,'EXISTS'|'OMITTED'])

 B
 ---------------------------

                B2C()
                - (<binární çíslo>)

                BITAND()
                - (<êetëzec1>,<êetëzec2>[,<vyplñující znak>])

                BITCHG()
                - (<êetëzec>,<poêadí bitu>)

                BITCLR()
                - (<êetëzec>,<poêadí bitu>)

                BITCOMP()
                - (<êetëzec1>,<êetëzec2>[,<vyplñující znak>])

                BITOR()
                - (<êetëzec1>,<êetëzec2>[,<vyplñující znak>])

                BITSET()
                - (<êetëzec>,<poêadí bitu>)
```

```
                BITTST()
                - (<êetëzec>,<poêadí bitu>)

                BITXOR()
                - (<êetëzec1>,<êetëzec2>[,<vyplñující znak>])

C
---------------------------

                C2B()
                - (<êetëzec>)

                C2D()
                - (<êetëzec>[,<poçet znakû>])

                C2X()
                - (<êetëzec>)

                CENTER()
                - (<êetëzec>,<délka>[,<vyplñující znak]>)

                CLOSE()
                - (<logický název souboru>)

                COMPARE()
                - (<êetëzec1>,<êetëzec2>[,<vyplñující znak>]

                COMPRESS()
                - (<êetëzec>[,<ruîené znaky>])

                COPIES()
                - (<êetëzec>,<poçet>)

D
---------------------------

                D2C()
                - (<celé çíslo>[,<délka>])

                D2X()
                - (<celé çíslo>[,<délka>])

                DATE()
                - ([<Volba>][,<datum>][,<formát>])

                DATATYPE()
                - (<êetëzec>[,<volba>])

                DELSTR()
                - (<êetëzec>,<çíslo>[,<délka>])

                DELWORD()
                - (<êetëzec>,<çíslo slova>[,<délka>])

                DIGITS()
                - ()
```

```
E
---------------------------

                EOF()
                - (<logický soubor>)

                ERRORTEXT()
                - (<èíslo chyby>)

                EXISTS()
                - (<jméno souboru>)

                EXPORT()
                - (<adresa>[,<êetëzec>][,<délka>][,<vyplòující znak>])

F
---------------------------

                FORM()
                - ()

                FIND()
                - (<êetëzec>,<fráze>)

                FREESPACE()
                - (<adresa>,<délka>)

                FUZZ()
                - ()

G
---------------------------

                GETCLIP()
                - (<jméno>)

                GETSPACE()
                - (<délka>)

H
---------------------------

                HASH()
                - (<êetëzec>)

I
---------------------------

                IMPORT()
                - (<adresa>[,<délka>])

                INDEX()
                - (<êetëzec>,<vzor>[,<poçátek>])

                INSERT()
                - (<nový ê.>,<starý ê.>[,<poçátek>][,<délka>][,<vyplòující znak ↩
                   >])
```

```
L
---------------------------

                LASTPOS()
                - (<vzor>,<êetëzec>[,<poçátek>])

                LEFT()
                - (<êetëzec>,<délka>[,<vyplñující znak>])

                LENGTH()
                - (<êetëzec>)

                LINES()
                - (<logický soubor>)

M
---------------------------

                MAX()
                - (<çíslo>,<çíslo>[,<çíslo>,...])

                MIN()
                - (<çíslo>,<çíslo>[,<çíslo>,...])

O
---------------------------

                OPEN()
                - (<logický název>,<nazev souboru>['APPEND'|'READ'|'WRITE'])

                OVERLAY()
                - (<nový ê.>,<starý ê.>[,<poçátek>][,<délka>][,<vyplñující znak>]

P
---------------------------

                POS()
                - (<vzor>,<êetëzec>[,<poçátek>])

                PRAGMA()
                - (<volba>[,<hodnota>])

R
---------------------------

                RANDOM()
                - ([<minimum>][,<maximum>][,<výchozí çíslo>])

                RANDU()
                - ([<výchozí çíslo>])

                READCH()
                - (<logický souborr>,<délka>)

                READLN()
                - (<logický soubor>)
```

```
                REMLIB()
                - (<název knihovny>)

                REVERSE()
                - (<êetëzec>)

                RIGHT()
                - (<êetëzec>,<délka>[,<vyplñující znak>])

S
----------------------------

                SEEK()
                - (<logický soubor>,<offset>[,{'BEGIN'|'CURRENT'|'END'}])

                SETCLIP()
                - (<jméno>[,<hodnota>])

                SHOW()
                - (<volba>[,<jméno>][,<oddëlující znak>])

                SIGN()
                - (<çíslo>)

                SOURCELINE()
                - ([<çíslo êádku>])

                SPACE()
                - (<êetëzec>,<poçet>[,<vyplñující znak>])

                STORAGE()
                - ([<adresa>][,<êetëzec>][,<délka>][<vyplñující znak>])

                STRIP()
                - (<êetëzec>[,{'B'|'L'|'T'}][,<ruîené znaky>])

                SUBSTR()
                - (<êetëzec>,<poçátek>[,<délka>][,<vyplñující znak>])

                SUBWORD()
                - (<êetëzec>,<çíslo slova>[,<délka (ve slovech)>])

                SYMBOL()
                - (<jméno>)

T
----------------------------

                TIME()
                - (<volba>)

                TRACE()
                - (<volba>)

                TRANSLATE()
```

```
                - (<êetëzec>[,<výstupní tab.>][,<vstupní tab.>][,<vyplñující zn ↩
                   .>])

                TRIM()
                - (<êetëzec>)

                TRUNC()
                - (<çíslo>[,<místa>])

U
---------------------------

                UPPER()
                - (<êetëzec>)

V
---------------------------

                VALUE()
                - (<jméno>)

                VERIFY()
                - (<êetëzec>,<seznam>,[,'MATCH'][,<poçátek>])

W
---------------------------

                WORD()
                - (<êetëzec>,<çíslo slova>)

                WORDINDEX()
                - (<êetëzec>,<çíslo slova>)

                WORDLENGTH()
                - (<êetëzec>,<çíslo slova>)

                WORDS()
                - (<êetëzec>)

                WRITECH()
                - (<logický soubor>,<êetëzec>)

                WRITELN()
                - (<logický soubor>,<êetëzec>)

X
---------------------------

                X2C()
                - (<êetëzec>)

                X2D()
                - (<hexa êetëzec>[,<délka>])

                XRANGE()
                - ([<start>][,<konec>])
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~
```

Ukázkový program
Konec pusté teorie...

```
-----------------------------------------------
 Funkce externí knihovny RexxSupport.library
```

Dûleùité informace
Knihovnu je têeba nejprve otevêít !!!

Abecední seznam Syntaxe

```
A
---------------------------
```

ALLOCMEM()
- (<délka>[,<atribut>])

```
B
---------------------------
```

BADDR()
- (<BCPL adresový êetëzec>)

```
C
---------------------------
```

CLOSEPORT()
- (<jméno>)

```
D
---------------------------
```

DELAY()
- (<çíslo>)

DELETE()
- (<soubor>)

```
F
---------------------------
```

FORBID()
- ()

FREEMEM()
- (<adresa>,<délka>)

```
G
---------------------------
```

GETARG()
- (<paket>[,<pozice>])

GETPKT()
- (<jméno>)

```
G
---------------------------

                NEXT()
                - (<adresa>[,<offset>])

                NULL()
                - ()

O
---------------------------

                OFFSET()
                - (<adresa>,<posun>)

                OPENPORT()
                - (<jméno>)

G
---------------------------

                PERMIT()
                - ()

R
---------------------------

                REPLY()
                - (<paket>,<rc>)

S
---------------------------

                SHOWDIR()
                - (<adresáê>[,'ALL'|'FILE'|'DIR'][,<oddëlující znak>])

                SHOWLIST()
                - (<volba>[,<jméno>][,<oddëlující znak>])

                STATEF()
                - (<název souboru>)

T
---------------------------

                TYPEPKT()
                - (<adresa>[,<mód>])

W
---------------------------

                WAITPKT()
                - (<jméno>)
```

1.85 Popis funkce ABBREV()

```
                ABBREV(<dlouhý êetëzec>,<krátky êetëzec>[,<délka>])
```

Tato funkce vrací booleovskou hodnotu, která udává, jestli je <krátký êetëzec> zkratkou êetëzce <dlouhý êetëzec>.

Nepovinným parametrem <délka> je moùno zadat minimální nutný poçet zhodných znakû, coù je implicitnë délka kratîího êetëzce.

```
Pêíklad:
SAY ABBREV('vollname','voll')   -> 1
SAY ABBREV('nahezu','nah',4)    -> 0
SAY ABBREV('libovolný','')      -> 1
```

1.86 Popis funkce ABS()

```
 ABS(<çíslo>)
```

Vrací absolutní hodnotu argumentu <çíslo>. To musí mít numerickou hodnotu.

```
Pêíklad:
SAY ABS(-5.35)     -> 5.35
SAY ABS(10)        -> 10
```

1.87 Popis funkce ADDLIB()

```
                ADDLIB(<jméno>,<priorita>[,<offset>,<verze>])
```

Pêipojí knihovnu funkcí nebo funkçní host do seznamu knihoven, spravovaným rezidentním procesem. Argument <jméno> oznaçuje buð jméno knihovny funkcí nebo funkçnímu hostu pêiêazený message-port. U tëchto jmen je rozliîováno mezi psaním velkých a malých písmen. Vîechny zadané knihovny by mëly stát v adresáêi "LIBS:".

Argument <priorita> urçuje prioritu pêi hledání a musí být celé çíslo z oblasti od 100 do -100. Argumenty <offset> a <verze> platí jen pro knihovny. Offset je celoçíselná vzdálenost od "query" vstupního bodu knihovny (obvykle se udáva -30). <verze> je rovnëù celé çíslo, které udává, jakou minimální verzi knihovny pouùít.

Funkce vrací booleovskou hodnotu, která informuje, zda byla operace úspëîná. Pokud jiù v ressource seznamu stejná poloùka existuje, bude vráceno 0.

Vîimnëte si prosím, ùe zadaná knihovna není v tomto okamùiku jeîtë otevêena, není ani testováno zda zadaná knihovna vûbec existuje. Knihovna bude otevêena aù bëhem hledání funkce. Arexxový interpret totiù pokud

nalezne funkci (pêíkaz), kterou nezná, prohledá kaùdou knihovnu v ressource seznamu podle jejich priorit a zjiîïuje zda nëkterá funkci nezná. Pokud se mu nëkterou knihovnu nepodaêí otevêít vrátí chybu çíslo
14
.

ARexx také nezkoumá, zda-li je port zadaného funkçního hostu otevêen.

Pêíklad:
```
SAY ADDLIB("Rexxsupport.library",0,-30,0)  -> 1

CALL ADDLIB "EtherNet",-30                 /* Gateway */
```

Knihovnu nebo funkçní host lze ze seznamu odstranit funkcí
REMLIB()
.

1.88 Popis funkce ADDRESS()

ADDRESS()

Vrací aktuální êetëzec adresy
hostu
. Adresa hostu je message-port, na který jsou posílány kommanda.

Pomocí funkce
SHOW()
mûùe být pêezkouîeno, zda potêebný externí host skuteçnë existuje.

Pêíklad:
```
SAY ADDRESS()    -> REXX
```

1.89 Popis funkce ARG()

ARG([<çíslo>][,'EXISTS'|'OMITTED'])

ARG() vrací poçet argumentû, které byly aktuálnímu prostêedí dány k dispozici. Je-li zadán argument <çíslo>, je vrácen odpovídající argumentový êetëzec. Pêi zadání çísla a klíçového slova EXISTS (existuje) nebo OMITTED (neexistuje) ukazuje booleovská vrácená hodnota stav odpovídajícího argumentu. Vîimnëte si prosím, ùe pêezkouîení klíçovým slovem EXISTS nebo OMITTED nemá za výsledek zjiîtëní, jestli má êetëzec nulovou hodnotu, ale jen jestli vûbec byl nëjaký zadán.

U klíçových slov staçí uvést první písmeno.

Pêíklad:
```
/*Vezmëme si argumenty: ('jedna',,10)*/
```

```
SAY ARG()          -> 3
SAY ARG(1)         -> jedna
SAY ARG(2,'O')     -> 1      /* Stací zadat první písmeno */
```

Podívejte se také na:
PARSE ARG

1.90 Popis funkce B2C()

B2C(<binární èíslo>)

Tato funkce pêemìní <binární èíslo> (tj. êetìzec sloùený pouze z 1 a 0) na odpovídající znak podle tabulky ASCII.

Mezery v êetìzci jsou pro pêehlednost pêípustné jen na hranicích jednotlivých bytû, tedy kaùdých osm èíslic.

Tato funkce se hodí obzvláîtë pro tvorbu êetìzcû, které mají být pouùity jako bitové masky nebo pêi ukládání bitových masek do pamëti

Inverzní funkcí k B2C() je
C2B()
.

```
Pêíklad:
SAY B2C('00110011')    -> 3
SAY B2C('01100001')    -> a
```

Podívejte se také na
C2D()

D2C()

C2X()

X2C()

D2X()

X2D()

1.91 Popis funkce BITAND()

BITAND(<êetìzec1>,<êetìzec2>[,<vyplñující znak>])

Funkce vrací textový êetìzec, který vznikl spojením argumentû <êetìzec1> a <êetìzec2>
logickým AND
, kde výsledek je tak dlouhý, jako delîí z obou operandových êetìzcû.

Je-li zadán <vyplòující znak>, je jím kratîí êetëzec vpravo vyplnën. Jinak je operace ukonçena na konci kratîího êetëzce. Zbytek delîího êetëzce je êipojen za výsledek.

Pêíklad:
```
BITAND('0313'x,'FFF0'x)    -> '0310'x /* sloucí dva hexadecimální êetëzce */
BITAND('A','J')            -> @
```

Více informací najdete u popisu funkce
BITCHG()
.

Podobné této funkci jsou také funkce:
BITCHG()

BITCLR()

BITCOMP()

BITOR()

BITSET()

BITTST()

BITXOR()

1.92 Popis funkce BITCHG()

BITCHG(<êetëzec>,<poêadí bitu>)

Zmëní stav bitû zadaných v argumentu <êetëzec>.

Pro pochopení vysvëtlím tento dëj tak, aby se i laik mohl pêesvëdçit o çinnosti této funkce.

Çíslo jenù je zadáno jako êetëzec (decimální,binární nebo hexadecimální) je pêeveden na binární çíslo (sloùené z jedniçek a nul). Pak je v tomto çísle zmënëna jedna çíslice (z 1 na 0 nebo 0 na 1), jenù je urçena çíselným parametrem <poêadí bitu>, který udává její pozici. Pêiçemù çíslice úplnë vpravo má pozici 0 (çísluje se zprava do leva).
Po této operaci je výsledný êetëzec pêeveden na znak podle ASCII tabulky.

Tímto zpûsobem jsou zpracovávány vîechny funkce jejichù název zaçína na "BIT" liîí se pouze v operaci provedené z binárním êetëzcem po pêevodu. Výsledek je u vîech tëchto funkcí textový êetëzec.

Pêíklad:
```
SAY BITCHG('d',0)        -> e
SAY BITCHG('0313'x,4)    -> po pêevedení na hexadecimální çíclo -> '0303'x
```

Pokud vám pêedcházející pêíklady nestaçili, podívejte se na program:

```
                Bitové manipulace
               .

   Podobné této funkci jsou také funkce:
                BITAND()

                BITCLR()

                BITCOMP()

                BITOR()

                BITSET()

                BITTST()

                BITXOR()
```

1.93 Popis funkce BITCLR()

```
                BITCLR(<êetëzec>,<poêadí bitu>)
```

Touto funkcí je zadaný bit v argumentu <êetëzec> smazán (vynulován). Parametr <poêadí bitu> udává pozici bitu jenù má být smazán. Bity se çíslují z prava do leva, pêiçemù bit úplnë vpravo má pozici 0.
Funkce vrací výsledek jako textový êetëzec. (znaky z ASCII tabulky)

Funkce funguje obdobnë jako funkce BITCHG(), pokud vám tedy není stále jasná funkce BITCLR(), doporuçuji vám prostudovat

```
                BITCHG()
               .
```

```
Pêíklad:
SAY BITCLR('0313'x,4)        -> '0303'x
SAY
                C2B
               (BITCLR('1010'b,1))   -> 00001000
```

Více informací najdete u popisu funkce

```
                BITCHG()
               .
```

Podobné této funkci jsou také funkce:

```
                BITAND()

                BITCHG()

                BITCOMP()

                BITOR()

                BITSET()

                BITTST()
```

BITXOR()

1.94 Popis funkce BITCOMP()

BITCOMP(<êetëzec1>,<êetëzec2>[,<vyplñující znak>])

Funkce porovnává argumenty <êetëzec1> a <êetëzec2> po bitech od bitu číslo 0. Vrácená hodnota je číslo prvního bitu, u kterého se êetëzce liîí, nebo -1, pokud jsou êetëzce identické. Pêiçemù bity se çíslují z prava do leva a zaçíná se nulou.

Pêíklad:
```
SAY BITCOMP('7F'x,'FF'x)          -> 7     /*Bit číslo 7*/
SAY BITCOMP('FF'x,'FF'x)          -> -1
SAY BITCOMP('01110'b,'01010'b)    -> 2
```

Více informací najdete u popisu funkce
BITCHG()
.

Podobné této funkci jsou také funkce:
BITAND()

BITCHG()

BITCLR()

BITOR()

BITSET()

BITTST()

BITXOR()

1.95 Popis funkce BITOR()

BITOR(<êetëzec1>,<êetëzec2>[,<vyplñující znak>])

Funkce vrací textový êetëzec, který vznikl spojením argumentû <êetëzec1> a <êetëzec2>
logickým OR
(nebo), kde výsledek je tak dlouhý,
jako delîí z obou operandových êetëzcû.

Je-li zadán <vyplñující znak>, je jím kratîí êetëzec vpravo vyplnën. Jinak je operace ukonçena na konci kratîího êetëzce. Zbytek delîího êetëzce je pêipojen za výsledek.

```
Pêíklad:
SAY BITOR('0313'x,'003F'x)        -> '033F'x
SAY C2B(BITOR('101'b,'011'b))     -> 00000111
```

Více informací najdete u popisu funkce
BITCHG()
.

Podobné této funkci jsou také funkce:
BITAND()

BITCHG()

BITCLR()

BITCOMP()

BITSET()

BITTST()

BITXOR()

1.96 Popis funkce BITSET()

BITSET(<êetëzec>,<poêadí bitu>)

Zmëní zadaný bit v argumentu <êetëzec> na 1. Bity se çíslují zprava do leva, pêiçemù se zaçíná nulou.

```
Pêíklady:
SAY BITSET('0313'x,2)            -> '0317'x
SAY
                C2B
               (BITSET('100000'b,1))    -> 00100010
```

Více informací najdete u popisu funkce
BITCHG()
.

Podobné této funkci jsou také funkce:
BITAND()

BITCHG()

BITCLR()

BITCOMP()

BITOR()

BITTST()

BITXOR()

1.97 Popis funkce BITTST()

```
                BITTST(<êetëzec>,<poêadí bitu>)
```

   Funkce vrací stav vybraného bytu v êetëzci <êetëzec>. Pozice zkoumaného bitu se urçuje argumentem <poêadí bitu> "êetëzec". Bity se çíslují zprava do leva, pêiçemù bit úplnë vpravo má pozici 0.

```
Pêíklad:
SAY BITTST('0313'x,4)    -> 1
SAY BITTST('11001'b,0)   -> 1   /* bit zcela vpravo má hodnotu 1 */
SAY BITTST('11001'b,1)   -> 0
```

   Více informací najdete u popisu funkce
                BITCHG()
               .

   Podobné této funkci jsou také funkce:
                BITAND()

                BITCHG()

                BITCLR()

                BITCOMP()

                BITOR()

                BITSET()

                BITTST()

1.98 Popis funkce BITXOR()

```
                BITXOR(<êetëzec1>,<êetëzec2>[,<vyplñující znak>])
```

   Funkce vrací textový êetëzec, který vznikl spojením argumentû <êetëzec1> a <êetëzec2>
                logickým exkluzívním OR
                (nebo), kde výsledek je
tak dlouhý, jako delîí z obou operandových êetëzcû.

   Je-li zadán <vyplñující znak>, je jím kratîí êetëzec vpravo vyplnën. Jinak je operace ukonçena na konci kratîího êetëzce. Zbytek delîího êetëzce je pêipojen za výsledek.

```
Pêíklad:
SAY BITXOR('0313'x,'001F'x)        -> '030C'x
```

Více informací najdete u popisu funkce
BITCHG()
.

Podobné této funkci jsou také funkce:
BITAND()

BITCHG()

BITCLR()

BITCOMP()

BITOR()

BITSET()

BITTST()

1.99 Popis funkce C2B()

C2B(<êetëzec>)

Funkce pêemëní textový <êetëzec> do odpovídajícího binárního êetëzce.

Tato funkce je inverzní k funkcí
B2C()
.

Pêíklad:
SAY SAY C2B('abc') -> 011000010110001001100011

Podívejte se také na
C2D()

D2C()

C2X()

X2C()

D2X()

X2D()

1.100 Popis funkce C2D()

C2D(<êetëzec>[,<poçet znakû>])

Pêemëní argument <êetëzec> z textových znakû na odpovídající desítkové

```
číslo (v ASCII znacích jako 0-9).

    Je-li  zadán  <poçet  zankû>, je z textového êetëzce brán pouze omezený
poçet  znakû,  který  je  pêeváðen  na çíslo.  Êetëzec je tedy useknut nebo
zleva vyplnën nulami. (Znaménkový bit je pro pêemënu rozîíêen.)

   K této funkci je inverzní
                D2C()
               .

Pêíklad:

SAY C2D('0020'x)         -> 32
SAY C2D('FFFF ffff'x)    -> -1
SAY C2D('FF0100'x,2)     -> 256

SAY C2D('A')             -> 65
SAY C2D('A',2)           -> 65
SAY C2D('AA')            -> 16705
SAY C2D('AA',1)          -> 65        /* je brán pouze první znak */


Podívejte se také na
                B2C()

                C2D()

                C2X()

                X2C()

                D2X()

                X2D()
```

1.101 Popis funkce C2X()

```
                C2X(<êetëzec>)

   Pêemëní  argument <êetëzec> z normálních textových znakû na odpovídající
hexadecimální çíslo (v ASCII znacích 0-9 a A-F).

   K této funkci je inverzní
                X2C()
               .

Pêíklad:
SAY C2X('abc')           -> 616263
SAY C2X('0'x)            -> 00
SAY C2X('41'x)           -> 41

Podívejte se také na
                C2D()
```

```
D2C()

B2C()

C2B()

D2X()

X2D()
```

1.102 Popis funkce CENTER()

```
 CENTER(<êetëzec>,<délka>[,<vyplñující znak]>)
```

Funkce centruje argument <êetëzec> uvnitê nového êetëzce jehoù délka je specifikovaná parametrem <délka>.

Je-li hodnota <délka> vëtîí neù délka êetëzce, jsou pêipojeny vyplñovací znaky nebo (standartnë) mezery. Pokud je ale zadaná <délka> menîí jsou krajní znaky êetëzce odêezány !

Z zouto funkcí je ekvivalentní (totoùná) funkce CENTRE().

Pêíklad:

```
SAY CENTER('abc',6)       -> ' abc  '
SAY CENTER('abc',6,'+')   -> '+abc++'
SAY CENTER('123456',3)    -> '234'
```

1.103 Popis funkce CLOSE()

```
CLOSE(<logický název souboru>)
```

Zavêe logický soubor, jehoù logické jméno bylo zadáno.

Byl-li soubor otevêen (funkcí OPEN()), udává booleovská hodnota úspëîné provedení operace.

Pêíklad:

```
CALL OPEN('input','Ram:pokus',R)
SAY CLOSE('input')      -> 1
```

nebo

```
CALL OPEN(OUT,'Ram:Zápis',W)
SAY CLOSE(OUT)
```

Blyùîí popis naleznete u funkce

OPEN

1.104 Popis funkce COMPARE()

```
COMPARE(<êetëzec1>,<êetëzec2>[,<vyplñující znak>]
```

Funkce porovnává dva êetëzce a vrací indexní hodnotu první pozice, kde se êetëzce liîí (nebo nula, kdyù jsou identické). Kratîí êetëzec je pêi potêebë vpravo vyplnën zadanými vyplñujícími znaky nebo (implicitnë) mezerami.

```
Pêíklad:
SAY COMPARE('abcde','abcce')      -> 4
SAY COMPARE('abcde','abcde')      -> 0
SAY COMPARE('abc++','abc+-','+')  -> 5
```

1.105 Popis funkce COMPRESS()

```
COMPRESS(<êetëzec>[,<ruîené znaky>])
```

Je-li argument <ruîené znaky> vypuîtën, odstraní funkce vîechny mezery z argumentu <êetëzce>. Je-li zadán volitelný argument <seznam> jsou likvidovány znaky uvedené v seznamu.

```
Pêíklad:
SAY COMPRESS('   Kol ik   ')       -> Kolik
SAY COMPRESS('++12-34-+','+-')     -> 1234
```

Podívejte se také na
STRIP()
a
TRIM()
.

1.106 Popis funkce COPIES()

```
COPIES(<êetëzec>,<poçet>)
```

Vytvoêí nový êetëzec tím zpûsobem, ùe zadaný poçet kopií originálního êetëzce spojí dohromady. Jako <poçet> mûùe být zadaná i nula. V tomto pêípadë je vrácen prázdný êetëzec.

```
Pêíklad:
SAY COPIES('abc',3)   -> abcabcabc
```

1.107 Popis funkce D2C()

```
                D2C(<celé číslo>[,<délka>])
```

Vytvoêí êetëzec, jehoù hodnota je znakovým (spakovaným) vyjádêením zadaného desítkového (decimálního) čísla. Parametrem <celé číslo> zadáváte celé kladné desítkové číslo které je pak pêavedeno na znak podle ASCII tabulky.

Pokud pêekroçíte rozsah ASCII tabulky tím, ùe napêíklad zadáte 256, poradí si s tím funkce velice zajímavë. Vrátí totiù opët první znak ASCII tabulky, pouze pêed nëj pêidá jedniçku. Pokud vîak pouùijete parametr <délka>, bude výsledný êetëzec zkrácen na poùadovanou délku odêíznutím levé části. Zadáte li jako délku 1, bude číslo 256 pêesnë odpovídat číslu 0.

K této funkci je inverzní

```
                C2D()
```

.

Více vám snad napoví tyto pêíklady:

```
Say D2C(65)            ->   A
Say D2c(32C)           ->   A    /* první znak ASCII tabulky se nezobazuje */
Say C2X( D2C(256) )    ->  0100
Say C2X( D2C(257) )    ->  0101
Say C2D( D2C(321) )    ->  321  /* Opaçná funkce k C2D */
Say C2D( D2C(321,1) )  ->  65   /* výsledek vnitêní funkce je zkrácen !!! */

SAY D2C(1097363306)    -> Ahoj  /* číslém lze vyjádêit i celé slovo */
SAY D2C(1097363306,2)  -> oj    /* dalîí ukázka zkrácení */
```

Podívejte se také na

```
                B2C()

                C2B()

                D2X()

                X2D()

                C2X()

                X2C()
```

1.108 Popis funkce D2X()

```
                D2X(<celé číslo>[,<délka>])
```

Funkce pêemnění desítkové (decimální) číslo na hexadecimální číslo, které pak mûùe být parametrem <délka> zkráceno na poùadovanou délku odêíznutím levé části.

Číslo musí být celé a kladné.

```
K této funkci je inverzní
                X2D()
               .

Pêíklad:
SAY D2X(31)         ->  1F
SAY D2X(255)        ->  FF
SAY D2X(255,1)      ->  F
SAY D2X(999999)     ->  F423F
SAY D2X(999999,2)   ->  3F        /* text byl zleva odêíznut na dva znaky */

Podívejte se také na
                C2D()

                D2C()

                B2C()

                C2B()

                C2X()

                X2C()
```

1.109 Popis funkce DATE()

```
                DATE([<volba>][,<datum>][,<formát>])
```

Funkce vrátí aktuální datum v zadaném formátu. Implicitní volba je 'NORMAL', která vrátí datum ve formátu DD MMM RRRR, napê. 20 APR 1992.

Dále je jako parametr <volba> moùné pouùít tato klíçová slova:

```
BASEDATE    Poçet dní od 1.ledna 0001
CENTURY     Poçet dní od 1.ledna 20. století
DAYS        Poçet dní od 1.ledna aktuálního roku.
EUROPEAN    Datum ve formátu DD/MM/RR
INTERNAL    Interní systémové dny. (poçet dnû od 1.1.1978)
JULIAN      Datum ve formátu RRDDD
MONTH       Aktuální mësíc (v malých a velkých písmenech)
NORMAL      Datum ve formátu DD MMM RRRR
ORDERED     Datum ve formátu RR/MM/DD
SORTED      Datum ve formátu RRRRMMDD
USA         Datum ve formátu MM/DD/RR
WEEKDAY     Den (v malých a velkých písmenech)
```

Tyto volby mohou být zkráceny. Stací zadání prvního znaku.

Funkce DATE() akceptuje také dalîí dva volitelné argumenty, s kterými mûùe být zadáno <datum> ve formë systémových dnû nebo v seêazené formë RRRRMMDD (RokyMësíceDny). Funkce DATE() pak toto datum konvertuje na datum jenù jste zadali jako parametr <Volba>.

Pêiçemù argument <datum> se zadávají systémové dny nebo seêazený datový formát. A argument <formát> pak specifikuje, který z tëchto formátû byl vlastnë pouùit. A mûùe být zadán buð 'I' nebo 'S'. Aktuální datum v systémových dnech pak mûùe být vyvoláno zadáním DATE('INTERNAL').

```
Pêíklad:
SAY DATE()       -> 14 Jul 1992      /* Vypíše datum ve formátu "NORMAL" */
SAY DATE('M')    -> July             /* Vypíše aktuální mësíc            */
SAY DATE(S)      -> 19920714         /* Vypíše datum ve formátu RRRRMMDD */
SAY DATE('W',19890609,'S')   -> Friday
 Následující pêíklad ukazuje vkládání funkce do funkce.
SAY DATE('S',DATE('I')+21)   -> 19920804
```

Podívejte setaké na
TIME()
, která pracuje zase s çasem.

1.110 Popis funkce DATATYPE()

DATATYPE(<êetëzec>[,<volba>])

Pomocí teto zajímavé funkce lze zjiîtovat jakého typu je parametr <êetëzec>. (çíslo, text s urçitými vlastnostmi, atd.) Není-li zadán druhý argument <volba>, funkce vratí NUM, pokud je êetëzec çíslo, jinak vrátí CHAR.

Jako parametr <volba> lze zadat tyto parametry: (stací zadat první písmeno)

ALPHANUMERIC	Abecední (A-Z, a-z, pêehlásky, ß,) a numerické (0-9) znaky
BINARY	Binární posloupnost cifer (binární çíslo)
LOWERCASE	Malá písmena (a-z, ù, é, $\times$)
MIXED	Velká a malá písmena
NUMERIC	Platná çísla
SYMBOL	Platné arexxové symboly (správný název promënné nebo êetëzce)
UPPER	Velká písmena (A-Z, ý, ú, ã, ß)
WHOLE	Celá çísla
X	Hexadecimální posloupnosti cifer

Funkce pak vrací
booleovský
výsledek, který informuje o tom,zda <êetëzec>
splñuje vámi zadané kritérium.

```
Pêíklad:
SAY DATATYPE('123')    -> NUM
SAY DATATYPE('1a f2','X')    -> 1
SAY DATATYPE('aBcde','L')    -> 0
```

1.111 Popis funkce DELSTR()

```
                DELSTR(<êetëzec>,<çíslo>[,<délka>])
```

Funkce vrací parametr <êetëzec> ze kterého vîak smaùe poçínaje pozicí <poçátek>, poçet znakû daný parametrem <délka>. Standardní délka je zbytek êetëzce.

```
Pêíklad:
SAY DELSTR('123456',2,3)     -> 156
```

Podívejte se také na
```
                DELWORD()
```
, která maùe celá slova.

1.112 Popis funkce DELWORD()

```
 DELWORD(<êetëzec>,<çíslo slova>[,<délka>])
```

Smaùe část argumentu <êetëzec>, který zaçíná zadaým slovem (<çíslo slova>) a je tvoêen z <délka> slov. Standardní délka je zbytek êetëzce. Ke zkrácenému êetëzci patêí i mezery za posledním slovem !

```
Pêíklad:
SAY DELWORD('êekni mi nëco pêíjemného',2,2)        -> 'êekni pêíjemného'
SAY DELWORD('jedna dva têi',3)                     -> 'jedna dva '
```

1.113 Popis funkce DIGITS()

```
                DIGITS()
```

Vrátí aktuální nastavení
```
                NUMERIC DIGITS
               .
```

```
Pêíklad:
NUMERIC DIGITS 6
SAY DIGITS()  -> 6
```

1.114 Popis funkce EOF()

```
                EOF(<logický soubor>)
```

Pêezkouší zadaný logický soubor a vrátí
```
                booleovskou hodnotu
                1 (pravda),
```
kdyù je dosaùeno konce souboru, jinak 0 (leù), pokud nebylo dosaùeno konce souboru.

```
Pêíklad:
SAY EOF(Jeden_soubor.exx)    -> 1

   S touto funkcí souvisí také
                OPEN()
                a
                CLOSE()
               .
```

1.115 Popis funkce ERRORTEXT()

```
                ERRORTEXT(<çíslo chyby>)
```

Vrátí chybovou hláîku, patêící zadanému arexxovému chybovému kódu. Není-li çíslo <çíslo chyby> platný chybový kód, je vrácen prázdný êetëzec.

```
Pêíklad:
SAY ERRORTEXT(41)     -> INVALID EXPRESSION

   Mohl by vás zajímat také
                seznam chybových hláîení
               .
```

1.116 Popis funkce EXISTS()

```
                EXISTS(<jméno souboru>)
```

Testuje, jestli externí soubor se zadaným jménem existuje. Jméno mûùe obsahovat název zaêízení, cestu k adresáêi a samozêejmë k souboru.

Cestu lze zadat i relativnë vzhledem k adresáêi ze kterého byl program spuîtën nebo který byl nastaven funkcí

```
                PRAGMA()
               .
```

```
Pêíklad:
SAY EXISTS('SYS:C/ED')   -> 1
SAY EXISTS('ENV:ENV')    -> 1
SAY EXISTS('ENV:Text')   -> 0
```

1.117 Popis funkce EXPORT()

```
                EXPORT(<adresa>[,<êetëzec>][,<délka>][,<vyplñující znak>])
```

Funkce kopíruje data z <êetëzce> do rezervované oblasti pamëti. Tato pamëïová oblast musí být zadána ve formë 4-bytové adresy, kterou vrací funkce

GETSPACE()
pêi êezervování pamëti. Argument <délka> oznaçuje maximální poçet znakû ke zkopírování. Standardní hodnotou je délka êetëzce. Je-li zadaná délka delîí neù êetezec, je zbývající oblast vyplnëna vyplñujícím znakem nebo (implicitnë) nulovými znaky ('00'x).

Vrácená hodnota udává poçet skuteçnë zkopírovaných znakû.

Pozor !!!
Touto funkcí mûùe být pêepsána libovolná systémová oblast, coù mûùe zpûsobit zhroucení systému. Je lepîí, pokud budete vùdy uvádët argument <délka>, nebo alespoñ testovat délku kopírovaného êetëzce, mûùe se totiù stát, ùe êetëzec bude díky chybë delîí neù vámi rezervovaný úsek pamëti.

Během kopírování je podchyceno stêídání taskû, takùe pêi kopírování dlouhých êetëzcû moùná klesne výkon systému.

Této funkci je podobná funkce
STORAGE()
, která vîak vrací pûvodní obsah pamëti.

Opaçná k této funkci je funkce
IMPORT()
.

Pêíklad:
```
poçet = EXPORT('0004 0000'x,'Odpovëð')
```

1.118 Popis funkce FORM()

FORM()

Vrací aktuální nastavení
NUMERIC FORM
.

Pêíklad:
```
NUMERIC FORM SCIENTIFIC
SAY FORM()   -> SCIENTIFIC
```

1.119 Popis funkce FIND()

FIND(<êetëzec>,<fráze>)

Funkce hledá frázi, která mûùe být tvoêena z víc slov, v êetëzci a vrací çíslo poçáteçního slova, kde se obë fráze shodují.

Pêíklad:
```
SAY FIND('Nyní je ças k akci','je ças')    -> 2
```

Podívej se také na
INDEX()

LASTPOS()

POS()

1.120 Popis funkce FREESPACE()

FREESPACE(<adresa>,<délka>)

Vrací pamëïový blok nastavené délky na interní pool interpretu. Argument <adresa> musí být 4 bytový êetëzec, který byl obdrùen pêi pêedchozím zavolání interní pêiêazovací funkce
GETSPACE()
.

Není vùdy potêeba interní reservované místo v pamëti uvolnit, protoùe se tak stane pêi ukonçení programu. Byl-li rezervován pêíliî velký kus pamëti, mohou pêi pêedávání na pool vzniknout problémy s pamëtí.

Vrácená hodnota je
booleovské hodnota
udávající úspëînosti dëje.

Pêíklad:
```
SAY FREESPACE('00042000'x,32)  -> 1
```

Podívejte setaké na:
ALLOCMEM()

FREEMEM()
Tyto funkce totiù slouùí pro práci se systemovou pamëtí.

1.121 Popis funkce FUZZ()

FUZZ()

Vrací aktuální nastavení
NUMERIC FUZZ
.

Pêíklad:
```
NUMERIC FUZZ 3
SAY FUZZ()   -> 3
```

1.122 Popis funkce GETCLIP()

```
                GETCLIP(<jméno>)

   Funkce prohledá
                clipboard
                a kdyù najde zápis odpovídající argumentu <jméno>, tak
vrátí  jemu  pêiêazenou hodnotu. Pêi porovnávání jmen není rozliîováno mezi
malými  a velkými písmeny. Kdyù není jméno v clipboardu nalezeno, je vrácen
prázdný êetezec.

Podívejte se také na
                SETCLIP()
               .

Pêíklad:
/*Pêedpokládáme, ùe 'çíslo' obsahuje 'PI=3.1415926'*/
SAY GETCLIP(çísla)   -> PI=3.1415926
```

1.123 Popis funkce GETSPACE()

```
                GETSPACE(<délka>)

   Rezervuje  pamëïový  blok  zadané  délky z interního poolu interpretu.
Vrácená  hodnota  je  4-bytová  adresa  rezervovaného bloku, který není ani
smazán  natoù  inicializován.  Jakmile arexxový program skonçí, je vnitêní
pamëï  pêedána  zpët  systému.  Proto  by  nemëla  být tato funkce pouùita
na rezervaci  pamëti  pro  externí programy.  Pamët lze opët uvolnit funkcí

                FREESPACE()
               ).

   Knihovna  REXXSupport.library obsahuje funkci
                ALLOCMEM()
               , která
rezervuje  pamëtový blok  z poolu  volné  systémové pamëti. Tato  funkce je
proto vhodná na rezervování pamëti pro externí programy.

Pêíklad:
SAY C2X(GETSPACE(32))    ->'0003BF40'x

   Pro práci s rezervovánou çástí pamëti slouùí funkce:
                IMPORT()

                EXPORT()
```

1.124 Popis funkce HASH()

```
 HASH(<êetëzec>)
```

Vrátí hashový atribut êetëzce jako desítkové çíslo a zaktualizuje interní hashovou hodnotu êetëzce.

Hashový atribut je çíslo pêídëlené êetëzci, které se çasto pouùívá v indexních rejstêících pro urychlení vyhledávání.

Hashová hodnota se získá tak, ùe se nejprve získá souçet decimálních ASCII hodnot vîech znakû v êetëzci, který se následnë celoçíselnë (//) podëlí 256. Následující procedûra napadobuje funkci HASH():

```
HASH: Procedure  /* Napodobenina funkce HASH() */
 Parse Arg text
  soucet = 0
  do i = 1 to length(text)
    soucet = soucet + c2d(substr(text,i,1))
  end
return soucet // 256
```

A po vyçerpávajícím výkladu následuje pár pêíkladû:

```
SAY HASH('1')       ->  49
SAY HASH('AMIGA')   ->  95
SAY HASH('MAGIA')   ->  95
SAY HASH('Amiga')   ->  223
```

1.125 Popis funkce IMPORT()

```
IMPORT(<adresa>[,<délka>])
```

Tato funkce tvoêí êetëzec zkopírováním dat z pamëti od zadané 4-bytové adresy. Není-li zadán argument <délka>, skonçí kopírování v okamùiku, kdy narazí na byte s nulovou hodnotou.

Pamët se rezervuje funkci

```
GETSPACE()
.
```

Podívejte se také na

```
EXPORT()
.
```

Pêíklad:

```
exthodn = IMPORT('0004 000'x,8)
```

1.126 Popis funkce INDEX()

```
INDEX(<êetëzec>,<vzor>[,<poçátek>])
```

Tato funkce slouùí pro hledání êetëzce <vzor> v êetëzci <êetëzec> a to poçínaje pozicí <poçátek>. Standardní poçáteçní pozice je 1. Není li hledaný êetëzec nalezen, vrací funkce 0, v opaçném pêípadë jeho pozici.

```
Pêíklad:
SAY INDEX("123456","23")   -> 2
SAY INDEX("123456","77")   -> 0
SAY INDEX("123123","23",3) -> 5

Podívej se také na
                FIND()

                LASTPOS()

                POS()
```

1.127 Popis funkce INSERT()

```
                INSERT(<nový ê.>,<starý ê.>[,<poçátek>][,<délka>][,<vyplñující ↩
                   znak>])
```

Tato funkce slouùí ke vloùení êetëzce <nový ê.> do <stary ê.>. Poçátek vkládání urçuje argument <poçátek> (implicitnë 0 - pêed starý êetëzec). V pêípadë, ùe <poçátek> je vëtîí neù délka strého êetëzce, bude <starý ê.> zprav doplnën zadaným znakem. Pokud uvedete parametr <délka>, bude zase <nový ê.> zkrácen nebo doplnën zadaným znakem.

```
Pêíklad:
SAY INSERT('ab','12345')                    -> ab12345
SAY INSERT('123','++',3,5,'-')              -> ++-123--
SAY INSERT('nový','starý êetëzec',0,10)     -> nový      starý êetëzec
SAY INSERT('nový','starý êetëzec',18)       -> starý êetëzec     nový

Podívejte se také na funkci
                OVERLAY()
               , která do starého êetëzce nevkládá,
ale pêepisuje jej.
```

1.128 Popis funkce LASTPOS()

```
                LASTPOS(<vzor>,<êetëzec>[,<poçátek>])
```

Funkce hledá pozpátku první výskyt zadaného vzoru v argumentu <êetëzec>, od zadané pozice (vzhledem k poçátku). Standardní pozice je konec (délka) êetëzce. Pokud je hledaný êetëzec nalezen je vrácená jeho pozice, v opaçém pêípadë 0.

```
Pêíklad:
SAY LASTPOS('2','1234')                  -> 2
SAY LASTPOS('2','1234234')               -> 5
SAY LASTPOS('2','123234',3)              -> 2
SAY LASTPOS('2','13579')                 -> 0
SAY LASTPOS('ab','ab fg df gh vf ab')    -> 16
SAY LASTPOS('ab','ab fg df gh vf ab',10) -> 1
```

Podívej se také na
FIND()

INDEX()

POS()

1.129 Popis funkce LEFT()

LEFT(<êetëzec>,<délka>[,<vyplñující znak>])

Tato funkce vrací levou çást argumentu <êetëzec> o zadané délce. Je-li êetëzec kratîí neù daná délka, je êetëzec zprava doplnën na zadanou delku poùadovaným znakem.

Pêíklad:
SAY LEFT('123456',3) -> 123
SAY LEFT('123456',8,'+') -> 123456++

1.130 Popis funkce LENGTH()

LENGTH(<êetëzec>)

Funkce vrátí délku êetëzce (poçet znakû).

Pêíklad:
SAY LENGTH('têi') -> 3

1.131 Popis funkce LINES()

LINES(<logický soubor>)

Funkce zjiîïuje kolik êádkû má zadaný <logický soubor>. Jako logický soubor lze pouùít také standartní datové proudy
STDOUT
,
STDIN
,
STDERR
.
Implicitnë funkce pouùívá STDIN.

Poznámka: Tato funkce je zvláîï vhodná programujete-li nëjaký pêekladový program. Protoùe pokud znáte celkový poçet êádkû,mûùete uùivatele informovat o tom, kolik procent je jiù hotovo.

Pêíklad:

```
 PUSH 'jeden êádek'
 PUSH 'jeîtë jeden'

                 SAY

                 LINES
                (STDIN)   -> 2

Podívejte se také na:
                 PUSH

                 QUEUE

                 PULL
```

1.132 Popis funkce MAX()

```
                 MAX(<çíslo>,<çíslo>[,<çíslo>,...])
```

Vrátí nejvyîîí hodnotu zadaného argumentu. Vîechny argumenty musí být numerické a musí být nejménë dva parametry zadány.

```
Pêíklad:
SAY MAX(2.1,3,-1)     -> 3

Podívej se také na
                 MIN()
```

1.133 Popis funkce MIN()

```
                 MIN(<çíslo>,<çíslo>[,<çíslo>,...])
```

Vrátí nejniùîí hodnotu zadaného argumentu. Vîechny argumenty musí být numerické a musí být nejménë dva parametry zadány.

```
Pêíklad:
SAY MIN(2.1,3,-1)     -> -1

Podívej se také na
                 MAX()
```

1.134 Popis funkce OPEN()

```
                 OPEN(<logický název>,<nazev souboru>[,'APPEND'|'READ'|'WRITE'])
```

Otevêe externí soubor pro zadanou operaci. Argument <soubor> definuje logické jméno, kterému je pêiêazen soubor. <Jméno souboru> je DOSové jméno

souboru. Cestu lze zadat i relativnë vzhledem k adresáêi ze kterého byl program spuîtën nebo který byl nastaven funkcí

PRAGMA()

. Poçet souçasnë otevêených souborû je teoreticky neomezen. Pêi opouîtëní programu jsou vîechny soubory automaticky zavêeny.

Funkce vrací booleovskou hodnotu,která nám êíká,zda byla operace úspëînë provedena.

Jako poslední parametr mûùete zadat danou operaci: (Staçí první písmeno.)

* APPEND - pêidávat nová data na konec souboru
* READ - çíst ze souboru
* WRITE - zapisovat (pûvodní soubor bude pêepsán)

Nikdy nelze otevêít jeden soubor, ze kterého chete çíst a zároveñ do nëj zapisovat ! Vyjímkou je napêíklad moùnost otevêení okna a pêesmërování výstupu a vstupu do nëj. Pêíklad jak tohle provést mûùete nalézt u funkce

PRAGMA()

, stejným zpûsob je pouùit v programu " Programy.rexx ", jenù zajiîtuje spuîtení programû pêímo z okna pêed vámi.

Pêíklad:

```
SAY OPEN ('MûjCon','CON:160/50/320/100/MûjCon/cds')   -> 1
SAY OPEN('Výstup','RAM:TEMP','W')   -> 1
```

Takto lze napêíklad zaêídít aby se text nevypisoval na obrazovku, ale do souboru:

```
CALL

CLOSE
(
STDOUT
)     /* Zavêení standartního výstupního souboru */
SAY OPEN(STDOUT,'Ram:text',W) -> 1   /* Pêesmërování výstupu do souboru */
```

1.135 Popis funkce OVERLAY()

OVERLAY(<nový ê.>,<starý ê.>[,<poçátek>][,<délka>][,<vyplñující znak>]

Tato funkce pêepisuje <starý ê.> novým. Pêepis zaçíná na pozici <poçátek>. Implicitní hodnota je 1. <nový ê.> je pêed vkládáním do nového êetëzce zkrácen nebo zprava doplnën zadaným znakem na délku urçenou parametrem <délka>.

Rozdíl oproti funkci

INSERT()

je jednoznaçný. Tato funkce starý

```
êetëzec pêepíîe.

Pêíklad:
SAY OVERLAY('bb','abcd')                          -> bbcd
SAY OVERLAY('4','123',5,5,'-')                    -> 123-4---
SAY OVERLAY('nový','starý êetëzec')               -> novýý êetëzec
SAY OVERLAY('nový','starý êetëzec',,6)            -> nový  êetëzec
SAY OVERLAY('pëkný','ty jsi hnusný jako noc',8)   -> ty jsi pëknýý jako noc
```

1.136 Popis funkce POS()

```
                POS(<vzor>,<êetëzec>[,<poçátek>])
```

Hledá první výskyt argumentu <vzor> v argumentu <êetëzec>, poçínaje pozici zadanou v argumentu <start>. Standardní pozice je 1.

Vrácená hodnota udává polohu hledaného ùetëzce. Pokud není êetëzec nalezen, funkce vrátí 0.

```
Pêíklad:
SAY POS('23','123234')     -> 2
SAY POS('77','123234')     -> 0
SAY POS('23','123234',3)   -> 4

Podívej se také na
                FIND()

                INDEX()

                LASTPOS()
```

1.137 Popis funkce PRAGMA()

```
                PRAGMA(<volba>[,<hodnota>])
```

Tato funkce umoùñuje programu zmënu rûzných atributû, které se vztahují na systémové prostêedí, kde je program provádën. Argument <volba> je klíçové slovo, které udává atribut, který má být mënën. Argument <hodnota> urçuje novou hodnotu atributu.

Funkcí vrácená hodnota závisí na atributu. Nëkteré vrátí pêedtím platnou hodnotu, jiné jen
booleovskou hodnotu,urçující selhání nebo úspëch operace.

Definovaná klíçová slova pro argument <volba> jsou:

```
DIRECTORY   Udává nový, aktuální adresáê. Aktuální adresáê se vztahuje
            k tëm jménûm souborû, které neobsahují explicitní údaj o
```

zaêízení. Vrácená hodnota je staré jméno adresáêe. PRAGMA('D') odpovídá PRAGMA('D',""). Tím je vrácena aktuální cesta, aniù by byla provedena zmëna adresáêe. Poznámka: V takto zadaném adresáêi budou napêíklad hledány externí funkce.

PRIORITY Udává novou prioritu tasku. Hodnota priority musí být çíslo z intervalu -128 aù 127, v praxi je tato oblast znaçnë zúùena. Arexxové programy by v ùádném pêípadë nemëly dostat vëtîí prioritu neù rezidentní proces, který je momentálnë prováděn s prioritou 4. Vrácená hodnota je pêedtím platná hodnota priority.

ID Vrátí identifikátor tasku (adresu taskového bloku) ve formë hexadecimálního 8-çíselného êetëzce. Identifikátor tasku je jednoznaçné oznaçení pr o volán í ARexxu a mûùe být pouùit pro generování jednoznaçného jména.

STACK Udává novou velikost zásobníku pro aktuální arexxový program. Je-li zmënëna velikost zásobníku, je vrácena pûvodní hodnota.

Implementované volby jsou:

PRAGMA('Window',{'NULL'|'WORKBENCH'})
- êídí pole WindowPtr tasku, çili zobrazování systemových requestrû jako napê. "Vloùte prosím zaêízení...". Pokud pouùijete argument 'NULL' (staçí 'N') budou potlaçena vîechna dialogová okna. Otevírání oken mûùete opët povolit volbou 'WORKBENCH' (staçí 'W'). Implicitnë je zobrazování oken povoleno.

PRAGMA('*'[,<jméno>])
- definuje zadané logické jméno jako aktuální handler (ovladaç) konzoly ("*"). Tím mûùe uùivatel otevêít dva datové proudy uvnitê jednoho. Je-li jméno vynecháno, je pouùito jména klientového procesu.

Pêíklad:
```
SAY PRAGMA('D','DF0:C')    -> Extras
SAY PRAGMA('D','DF1:C')    -> Workbench
SAY PRAGMA('PRIORITY',-5)  -> 0
SAY PRAGMA('ID')           -> 00221ABC
CALL PRAGMA '*',
                STDOUT
               SAY PRAGMA("STACK",8092)   -> 4000
```

Spouîtíte-li arexxový program napêíklad z MultiView asi brzy zjistíte, ùe pêíkaz

```
SAY
```

je naprosto neúçinný. Je tomu proto, ùe neexistuje datový proud

```
STDOUT
```

(ani STDIN) na který jsou smëêována data pêi pouùití pêíkazu SAY. Tento datový proud lze jednosuîe otevêít funkcí

```
OPEN()
```

. Pokud vîak potêebujete jako vstup i výstup jedno okno shellu, musíte to provést

napêíklad takto:

```
  /* Otevêení výstupu a vstupu do okna shellu (zaêízení CON:)
  Call Open(STDOUT,'CON:',W)
  Call Pragma '*',
                 STDOUT
                  call Open(
                 STDIN
                ,'*',R)
```

Pokud nevíte jak zacházet se zaêízením CON:, mûùu vám poskytnout dokumentaci k jedné skvëlé náhradë Shellu, která vîak podporuje mnohem více parametrû neù klasický Shell. Jedná se o KingCON .

Dalîí informace naleznete v
Poznámce ke spuîtëní ukázkových programû
.

1.138 Popis funkce RANDOM()

RANDOM([<minimum>][,<maximum>][,<výchozí çíslo>])

Funkce vrátí celoçíselné pseudonáhodné çíslo z oblasti <minimum> aù <maximum>. Implicitní hodnoty jsou <0,999>. Interval min-max musí být menîí nebo roven 1000. Je-li potêeba vëtîích hodnot, je moùné výsledky funkce RANDU() odpovídajícím zpûsobem zvëtîit. Argument <výchozí çíslo> mûùe být zadán na inicializaci interního stavu generátoru náhodných çísel.

Poznámka:
Funkce RANDOM() negeneruje úplnë náhodná. Jsou totiù vytváêena pomocí znaçnë sloùitého algoritmu. Výpoçet náhodného çísla je vîak provádën od urçitého základu (çísla). (Mûùeme se setkat s anglickým názvem "seed".) A vy máte moùnost tento základ zmënit. Tím zajistíte, ùe posloupnost náhodných çísel bude vùdy jiná. Pokud tedy jako argument <výchozí çíslo> zadáte funkci
TIME()
s parametrem SECOND, çili TIME(S), dostanete vùdy jiné çíslo které zvíîí náhodnost funkce RANDOM().

Pokud se vám tento kok zdá pêíliî sloùitý, mûùete jej vynechat, ale budete si muset zvyknout na to, ùe program si bude vymýîlet po kaùdém spuîtëní stejná çísla.

Pêíklad:

```
SAY RANDOM(1,6)             /* Mûùe být jedna  */
SAY RANDOM(1,6)            /* zase jiné çíslo */
SAY RANDOM(1,6,TIME(S))   /* Tohle je teprve náhhodné çíslo ! */
```

Podívejte se také na
RANDU()
.

1.139 Popis funkce RANDU()

```
RANDU([<výchozí číslo>])
```

Vrátí rovnomërnë rozdëlené pseudonáhodné çíslo z oblasti 0 aù 1. Poçet desetinných míst je roven nastavení

```
NUMERIC DIGITS
```

. RANDU() mûùe být pouùito na generování çísel z libovolného intervalu (násobením, dëlením, mocnëním... výsledku).

Volitelný argument <výchozí çíslo> mûùe být zadán na inicializaci interního stavu generátoru náhodných çísel.

Pêíklad:

```
SAY RANDU()             -> 0.312520045
NUMERIC DIGITS 3
SAY RANDU()             -> 0.873
```

Podívejte se také na

```
RANDOM()
```

kde najdete inforamci o tom, jak je pseudonáhodné çíslo naprosto nenáhodné.

1.140 Popis funkce READCH()

```
READCH(<logický soubor>,<délka>)
```

Naçte zadaný poçet znakû ze zadaného logického souboru do êetëzce. Délka vráceného êetëzce odpovídá poçtu skuteçnë pêeçtených znakû a mûùe být menîí neù poùadovaná délka, napê. bylo-li dosaùeno konce souboru.

Pêíklad:

```
 text = READCH('VSTUP',10)    /* naçte 10 znakû */
```

Obdobná je funkce

```
READLN()
```

.

1.141 Popis funkce READLN()

```
READLN(<logický soubor>)
```

Naçte jeden êádek ze zadaného logického souboru do êetëzce.

Vrácený êetëzec neobsahuje znak pro posun êádkû.

Pêíklad:

```
 êádek = READLN('Mûjsoubor')
```

Obdobná je funkce
READCH()
.

1.142 Popis funkce REMLIB()

REMLIB(<název knihovny>)

Odstraní zápis se zadaným jménem ze
seznamu knihoven
, který je
spravován residentním procesem. Je-li nalezen zápis a úspëînë odstranën, je vrácena booleovská hodnota 1.

Tato funkce nerozliîuje mezi knihovnami funkcí a funkçními hosty, ale jednoduîe odstraní zadaný zápis.

Pêíklad:
```
SAY REMLIB('Mojeknihovna.library')   -> 0  /* Bohuùel taková neexistuje */
```

Podívejte se také
ADDLIB()
.

1.143 Popis funkce REVERSE()

REVERSE(<êetëzec>)

Funkce obrátí poêadí znakû v êetëzci.

Pêíklad:
```
SAY REVERSE('?en çorP')    -> Proç ne?
```

1.144 Popis funkce RIGHT()

RIGHT(<êetëzec>,<délka>[,<vyplñující znak>])

Tato funkce vrátí êetëzec o délce <délka> odêíznutím zprava z argumentu <êetëzec>. Je-li vrácená çást êetëzce menîí neù poùadovaná délka, je zleva doplnëna a to vyplñujícím znakem nebo (imlicitnë) mezerami.

Pêíklad:
```
SAY RIGHT('123456',4)       -> 3456
SAY RIGHT('123456',8,'+')   -> ++123456
```

1.145 Popis funkce SEEK()

```
                SEEK(<logický soubor>,<offset>[,{'BEGIN'|'CURRENT'|'END'}])
```

Tato funkce slouùí pro posun v logickém souboru, který byl pêedem otevêen funkcí

```
                OPEN()
               .
```

Posun v souboru se zadává parametrem <offset>. Têetí nepovinný parametr udáva zda bude posun proveden vzhledem k zaçátku ('BEGIN'), konci ('END') nebo k souçasné pozici. Pokud têetí parametr vynecháte, funkce se bude chovat jako by jste zadali 'CURRENT'.

Vrácená hodnota je nová pozice vzhledem k poçátku souboru.

Tato funkce bohuùel neumí posun po êádcích, který je çasto potêebnëjîí, ale vy si snad nëjak poradíte. (Napêíklad s pomocí

```
                LENGTH()
               .
```

```
Pêíklad:
SAY SEEK('Vstup',10,'B')      -> 10
SAY SEEK('Vstup',0,'E')       -> 356  /*Délka souboru*/
```

1.146 Popis funkce SETCLIP()

```
                SETCLIP(<jméno>[,<hodnota>])
```

Uloùí do

```
                clipboardu
```

nový pár <jméno>,<hodnota>, který je spravován rezidentním programem. Existuje-li zápis s tímto jménem, je jeho hodnota zadanou hodnotou aktualizována (pêepsána). Zápisy mohou být odstranëny zadáním nulové hodnoty (ùádný argument).

Funkce vrací

```
                booleovskou hodnotu
               , oznaçující úspëch çi neúspëch operace.
```

```
Pêíklad:
SAY SETCLIP('Cesta','DF0:s')    -> 1
SAY SETCLIP('Cesta')            -> 1
```

Podívejte se také na

```
                GETCLIP()
               .
```

1.147 Popis funkce SHOW()

```
                SHOW(<volba>[,<jméno>][,<oddëlující znak>])
```

Vrátí seznam z resource seznamu specifikovaného parametrem <volba> nebo pêezkouší, jestli ve vybraném seznamu exisruje záznam se zadaným jménem.

Jako <volba> lze zadat tato klíçová slova: (stací první písmeno)

```
CLIP        -   Pêezkouší jména v clipboardu
FILES       -   Pêezkouší jména momentálnë otevêených souborû
LIBRARIES   -   Pêezkouší jména v seznamu knihoven
PORTS       -   Pêezkouší jména v seznamu systémových portû
```

Je-li argument <jméno> vynechán, vrátí funkce êetëzec s ressource jmény, oddëleno mezerou nebo <vyplñujícím znakem>. Je-li zadáno <jméno>, urçuje vrácená booleovská hodnota, jestli bylo jméno v ressource seznamu nalezeno.

Ve jménech poloùek v seznamu se rozliíuje mezi velkými a malými písmeny !

```
Pêíklad:
SAY SHOW(P,,',')    ->   ... MULTIVIEW.1 MULTIVIEW.1.1 ... a dalíí
SAY SHOW(L,'rexxsupport.library') -> 1  /* Pozorn na velikost písmen ! */
SAY SHOW(L,'RexxSupport.library') -> 0  /* Sranda, co ?                */

Podívejte se také na
                SHOWLIST()
               .
```

1.148 Popis funkce SIGN()

```
 SIGN(<çíslo>)
```

Funkce vrací 1 pokud je hodnota argumentu <çíslo> kladná, 0 pokud je argument nulový nebo -1 v pêípadë, ùe je argument záporný.

Argument musí být vùdy numerický !

```
Pêíklad:
SAY SIGN(12)     ->  1
SAY SIGN(-33)    -> -1
SAY SIGN(5*5-25) ->  0
```

1.149 Popis funkce SOURCELINE()

```
                SOURCELINE([<çíslo êádku>])
```

Vrátí text zadané êádky právë bëùícího arexxového programu. Je-li argument êádek vypuîtën, vrátí funkce celkový poçet êádek souboru.

Tato funkce je çasto pouùívána pro çtení dat pêímo s programu. Problémem je ovîem zjistit na kterém êádkû data zaçínají. Jednu z moùností

```
vám blíùe
                popíîu
               .
```

Upozornëní:
Pokud bude program spuîtën s adresáêe s pêíliî dlouhou pêístupovou cestou, vrátí funkce pouze prázdný êetëzec, coù mûùe zpûsobit pëknou neplechu. Podle mých testû nesmí být délka pêístupové cesty delîí neù 25 znakû a to vçetnë lomítek, dvojteçky a celého názvu zaêízení (Ram Disk:). Délku si mûùete ovëêit napêíklad takto: ªLength(Pragma('D'))°, aktuální adresáê vîak musí souhlasit s adresáêem z nëhoù byl program spuîtën.

Pêíklad:

```
/*Jednoduchý testovací program*/
SAY SOURCELINE()   -> 3)
SAY SOURCELINE(1)  -> /*Jednoduchý testovací program*/
```

Poçet êádkû jakéhokoliv souboru lze zjistit funkcí

```
               LINES()
              .
```

1.150 Popis funkce SPACE()

```
               SPACE(<êetëzec>,<poçet>[,<vyplñující znak>])
```

Formátuje argument <êetëzec> tak, ùe mezi slova vloùí zadaný <poçet> mezer nebo jiný <vyplñující znak>. Pokud zadáte jako <poçet> 0, budou mezery odstranëny.

Pêíklad:

```
SAY SPACE('Nyní je ças',3)    -> 'Nyní   je   ças'
SAY SPACE('Nyní je ças',0)    -> 'Nyníjeças'
SAY SPACE('1 2 3',1,'+')      -> '1+2+3'
```

Mezery z êetezce lze také odstanit funkcí

```
               STRIP()
               nebo
               COMPRESS()
              .
```

1.151 Popis funkce STORAGE()

```
               STORAGE([<adresa>][,<êetëzec>][,<délka>][<vyplñující znak>])
```

Funkce STORAGE() bez argumentû vrátí kapacitu volné pamëti. Je-li zadána adresa, musí být zadána jako 4-bytový êetëzec. Funkce zkopíruje data z êetëzce na danou adresu. Parametr "délka" oznaçuje maximální poçet kopírovaných bytû a má standardnë délku êetëzce. Je-li zadaná délka vëtîí

neù sám êetëzec, je zbytek vyplnën nulovými byty nebo vyplñujícím znakem.

Vrácená hodnota oznaçuje pêedchozí obsah pamëïové oblasti. Tento mûùe být pozdëji pouùit k obnovení pûvodního obsahu.

Pozor !!!
Touto dunkcí mûùe být pêepsána libovolná systémová oblast, coù mûùe zpûsobit zhroucení systému. Je lepîí, pokud budete vùdy uvádët argument <délka>, nebo alespoñ testovat délku kopírovaného êetëzce, mûùe se totiù stát, ùe êetëzec bude díky chybë delîí neù vámi rezervovaný úsek pamëti.

```
Pêíklad:
SAY STORAGE()    ->  248400
StaryObsah = STORAGE('0004 000'x,'Odpovëð')
CALL STORAGE '0004 000'x,,32,'+'
```

Této funkci je podobná funkce
EXPORT()
, která vîak vrátí délku zkopírovaného êetëzce.

1.152 Popis funkce STRIP()

```
STRIP(<êetëzec>[,{'B'|'L'|'T'}][,<ruîené znaky>])
```

Tato funkce odstrañuje z argumentu <êetëzec> mezery nebo jiné <ruîené znaky>. Druhým argumentem mûùete specifikovat které mezery mají být odstranëny:

```
 Trailing  -  koncové
 Leading   -  poçáteçní
 Both      -  obojí  (implicitní volba)

Pêíklad:
SAY STRIP('  Jak prosím?  ')      -> Jak prosím?
SAY STRIP('  Jak prosím?  ','L')  -> Jak prosím?
SAY STRIP('++123+++','B','+')     -> 123
SAY STRIP('>> A hoj ¶+','+¶> ')   -> A hoj
SAY STRIP('>> A hoj ¶+','+¶>')    ->  A hoj  /* Poçáteçní mezera nebude */
                                             /* odstranëna.             */
```

Mezery z êetezce lze také odstanit funkcí
COMPRESS()
nebo
TRIM()
.

1.153 Popis funkce SUBSTR()

```
SUBSTR(<êetëzec>,<poçátek>[,<délka>][,<vyplñující znak>])
```

Vrátí çást êetëzce, která zaçíná na zadané pozici a má zadanou délku.

Poçáteçní pozice <poçátek> musí být kladné çíslo. Jako standardní <délka> platí zbytek êetëzce. Je-li výsledný êetëzec kratîí neù zadaná délka, je zprava vyplnën mezerami nabo vyplñujícím znakem.

Pêíklad:
```
 SAY SUBSTR('123456',4,2)   -> 45
 SAY SUBSTR('mojejméno',5,6,'=')  -> jméno=
```

Podívejte se také na
SUBWORD()
.

1.154 Popis funkce SUBWORD()

```
 SUBWORD(<êetëzec>,<çíslo slova>[,<délka (ve slovech)>])
```

Vrátí çást êetëzce, poçínaje daným slovem o délce <délka>, která udává poçet slov výsledného êetëzce. Jako standardní délka platí zbývající délka êetëzce.

Vrácený êetëzec nemá v ùádném pêípadë mezery na poçátku a na konci.

Pêíklad:
```
 SAY '>'SUBWORD('Nyní je ças  ',2,2)'<'   -> >je ças<
```

1.155 Popis funkce SYMBOL()

```
SYMBOL(<jméno>)
```

Pêezkouîí, jestli je argument <jméno> platný arexxový symbol (promënná).

* Není-li platný symbol (obsahuje nepovolené znaky), vrátí funkce BAD.
* Je-li symbol platný, ale není inicializován, vrátí funkce LIT.
* Je-li symbol platný a je mu pêiêazena hodnota, je vráceno VAR.

Poznámka: Platný arexxový symbol mûùe obsahovat znaky: A-Z,a-z,$,_,!,@,#, popêípadë teçku pro tvorbu
polí promenných
(Sloùených symbolû).

Pêíklad:
```
SAY SYMBOL('J')    -> VAR
SAY SYMBOL('x')    -> LIT
SAY SYMBOL('++')   -> BAD
```

1.156 Popis funkce TIME()

```
 TIME(<volba>)

   Vrátí  aktuální  systémový  ças nebo êídí interní poçítání çasu.

   Platná klíçová slova jsou: (staçí první písmeno)

 CIVIL        Vrátí aktuální ças v 12 hodinovém formátu. Hodina/minuta
 HOURS        Vrátí aktuální ças v hodinách od pûlnoci.
 MINUTES      Vrátí aktuální ças v minutách od pûlnoci.
 NORMAL       Vrátí aktuální ças v 24 hodinovém formátu.
 SECONDS      Vrátí aktuální ças v sekundách od pûlnoci.

   Není-li  zadána  volba,  vrátí  funkce aktuální systémový ças ve formátu
HH:MM:SS. Stejnë jako volba 'NORMAL'

   Následující klíçová slova pracují s interním çasovým registrem:

 RESET        Vrátí stav poçítadla v sekundách a vynuluje jej.
 ELAPSED      Pouze vrátí stav poçítadla v sekundách.

   Po  spuîtëní  programu  poçítadlo  stojí.  Pokud  tedy interpret narazí
na nëkterou  s  funkcí TIME('R') nebo TIME('E') poprvé a to kdekoliv, têeba
i v podmínce, vrátí hodnotu 00.0 a spustí çasový registr (poçítadlo).

   Poçítadlo udává ças vùdy v sekundách s pêesností dvou desetinných míst.

Pêíklad:
 /*Dejme tomu, ùe je 1:02 v noci...*/
 SAY TIME('C')     -> 1:02 AM
 SAY TIME('HOURS')    -> 1
 SAY TIME('M')     -> 62
 SAY TIME('S')     -> 3720
 SAY TIME('N')     -> 01:02:54
 SAY TIME()        -> 01:02:00

 /* Mëêení çasu - TP */
 TIME(R)
 DO FOR 10
   SAY TIME(E)
 END

To je ale rychlost, ùe ?
```

1.157 Popis funkce TRACE()

```
                TRACE(<volba>)

   Nastaví
                monitorovací
                 reùim  podle klíçového slova <volba>.

   Na  to  musí  být  pouùit jedna z povolených abecedních nebo prefixových
voleb.   Funkce   TRACE()   mëní   monitorování   i   bëhem  interaktivního
monitorování.  V  tomto  pêípadë  jsou pêíkazy TRACE ve zdrojovém programu
```

ignorovány. Vrácená hodnota je pêedchozí aktivní mód.

Pouùívání této funkce vyùaduje znalosti monitorování, které je spolu s dalîími vëcmi popsáno v
6. kapitole
.

1.158 Popis funkce TRANSLATE()

TRANSLATE(<êetëzec>[,<výstupní tab.>][,<vstupní tab.>][,< ↩
vyplñující zn.>])

Tato funkce vytvoêí pêekladovou tabulku a na základë této tabulky nahradí vybrané znaky z argumentu <êetëzec>. Pêekladová tabulka je zadána dvëma êetëzci <vstupní tab.> a <vstupní tab.>. Pêi pêekladu je kaùdý znak z êetëzce, který je obsaùen ve vstupní tabulce zmënën na odpovídající znak ve výstupní tabulce.

Pokud je výstupní tabulka kratîí neù vsupní, je doplnëna mezerami nebo vyplñujícím znakem (pokud je zadán). Výsledný êetëzec je stejnë dlouhý jako pûvodní êetëzec. Vstupní a výstupní tabulky mohou být libovolnë dlouhé.

Pokud zadáte pouze první parametr (bez tabulek), bude <êetëzec> je pêemënën na velká písmena. Pouhou konverzi na velká písmena vîak umí rychleji funkce
UPPER()
.

```
Pêíklad:
 SAY TRANSLATE("abcde","123","cbade","+")   ->  321++
 SAY TRANSLATE("malý")                      ->  MALÝ /* Poznámka: Norma ATO-E2 */
 SAY TRANSLATE("0110","10","01")            ->  1001
 SAY TRANSLATE('MÄß se dobÓe ?','aîê','ÄßÓ') -> Maî se dobêe ?
```

1.159 Popis funkce TRIM()

TRIM(<êetëzec>)

Odstraní koncové mezery z argumentu <êetëzec>.

Tato funkce je výsledkem identická s funkcí:
STRIP
(<êetëzec>,'T')

```
Pêíklad:
 SAY LENGTH(TRIM(' abc    '))     -> 4
```

Mezery z êetezce lze také odstanit funkcí

```
COMPRESS()
nebo
STRIP()
.
```

1.160 Popis funkce TRUNC()

```
TRUNC(<èíslo>[,<místa>])
```

Vrátí celoèíselnou èást argumentu <èíslo>, následovanou daným poètem desetinných míst.

Standartní hodnota pro argument <místa> je 0. èíslo je v pøípadì potøeby doplnìno nulami.

Tato funkce èíslo nezaokrouhluje, pouze oøezává desetinná místa !

Poznámka: Tato funkce bez druhého parametru nahrazuje INT(). Matematik zajisté ví, o co se jedná...

Pøíklad:

```
SAY TRUNC(123.456)   -> 123
SAY TRUNC(123.456,4) -> 123.4560
```

1.161 Popis funkce UPPER()

```
UPPER(<øetìzec>)
```

Funkce pøemìní øetìzec na velká písmena.

Tato funkce je výsledkem identická s funkcí TRANSLATE (<øetìzec>), jen je pøi kratších øetìzcích ponìkud rychlejší.

Pokud pouívate kódování ATO-E2 funguje tato funkce korektnì i pro~písmena s èeskou diakritikou. Pokud pouíváte napøíklad KOI8, musíte si pomoci takto:

```
SAY TRANSLATE(UPPER(øetezec),'ÁÈÏÉËÍÑÓØÒÙÚÛ','áèïéëíñóøòùúû')
```

Pøíklad:

```
SAY UPPER('Pìkný den')    -> PËKNÝ DEN
```

Poznámka: Pokud pouíváte KOI8, funkce vám vrátí -> PëKNý DEN.

1.162 Popis funkce VALUE()

```
 VALUE(<jméno>)

   Vrátí hodnotu symbolu, který pêedstavuje argument <jméno>.

  * Pokud promënná (symbol) není nadefinována vrátí zpët argument <jméno>
    sloùený  z velkých písmen.

  * Pokud  argument  <jméno> bude obsahovat znaky, které symboly obsahovat
    nemohou, dojde k chybë.  (Povolené znaky:  "A-Z,a-z,$,_,!,@,#,.".)

   K lepîimu pochopení snad pêispëje tento pêíklad:

/* Ukázka funkce VALUE() - 1 - TP */
 Name='Lenka' ; Lenka='Vlasty'
 SAY Name 'je dcera od' VALUE(Name)'.'  -> Lenka je dcera od Vlasty.

   Lze vîak realizovat i sloùitëjîí operace:

/* Ukázka funkce VALUE() - 2 - TP */
/* Tvorba databáze: */
 Jan_M_S   = 'neznámé'
 Lenka_Z_R = '651 21 36'
 /* Co çíst ? */
 Jmeno = 'Jan' ; Pohlavi = 'M' ; Stav = 'S'
 Say Jmeno 'má telefonní çíslo:' VALUE(Jmeno'_'Pohlavi'_'Stav)

 Program vypíše -> Jan má telefonní çíslo: neznámé

 Zkuste 6 êádek zmënit na: Jmeno = 'Lenka' ; Pohlavi = 'Z' ; Stav = 'R'

   Snad je vám jiù význam této funkce jasný.
```

1.163 Popis funkce VERIFY()

```
 VERIFY(<êetëzec>,<seznam>,['MATCH'][,<poçátek>])

   Vrátí  indexní  pozici  prvního  znaku v argumentu <êetëzec>, který není
v êetëzci  <seznam>  nebo  0,  kdyù  jsou  vîechny  znaky v seznamu.

   Je-li pouùito klíçové slovo 'MATCH', vrátí funkce indexní pozici prvního
znaku  v argumentu  <êetëzec>, který je obsaùen v êetëzci <seznam>, nebo 0,
kdyù ùádný ze znakû není na seznamu.

   V  obou pêedchozích pêípadech probíhá verifikace (kontrola) aù od pozice
dané parametrem <poçátek>. Implicitní hodnota je 1.

Pêíklad:
SAY VERIFY('123456','0123456789')     -> 0  /* Obsahuje pouze çísla ? */
SAY VERIFY('123a56','0123456789')     -> 4  /* Obsahuje ? Neobsahuje. */
SAY VERIFY('123a45','abcdefghij','M')   -> 4  /* Zakázané znaky         */
```

1.164 Popis funkce WORD()

```
 WORD(<êetëzec>,<çíslo slova>)
```

   Vrátí vybrané <çíslo slova> z argumentu <êetëzec>, nebo prázdný êetëzec, kdyù má êetëzec ménë slov neù které ùádáte.

   Vrácené slovo je bez úvodních a koncových mezer.

```
Pêíklad:
 SAY '>'WORD('Nyní je jiù ças   ',2)'<'   -> >je<
```

1.165 Popis funkce WORDINDEX()

```
 WORDINDEX(<êetëzec>,<çíslo slova>)
```

   Vrátí poçáteçní pozici slova urçeného argumentem <çíslo slova> z argumentu <êetëzec>, nebo 0, kdyù má êetëzec ménë slov neù které zadáte.

   Vrácená pozice je ve znacích od poçátku êetëzce.

```
Pêíklad:
 SAY WORDINDEX('Nyní je jiù ças   ',3)    -> 9
```

1.166 Popis funkce WORDLENGTH()

```
 WORDLENGTH(<êetëzec>,<çíslo slova>)
```

  Vrátí délku vybraného slova z argumentu <êetëzec>.

```
Pêíklad:
 SAY WORDLENGTH('çtyêi pët îest',3)  -> 4
 SAY WORDLENGTH('Strom,kûñ,kaña',1) -> 15 /* Pozor na to !!! */
```

1.167 Popis funkce WORDS()

```
                WORDS(<êetëzec>)
```

   Vrátí poçet slov v argumentu <êetëzec>.

   Jako oddëlovaçe slov jsou brány pouze mezery, takùe slova oddëlená napê. pouze çárkou jsou brána jako jedno slovo. Pokud chcete pêeto zjistit poçet slov mûùete pouùít funkci napê funkci

```
                TRANSLATE()
               , kterou odstraníte çárky
```

nebo jiné znaky. Lze vîak pouùít také funkci

```
                COMPRESS()
                nebo
```

```
                STRIP()
               .

Pêíklad:
 SAY WORDS("Jak se ti daêí?')                      -> 4
 SAY WORDS('Strom,kûñ,kañka',1)                    -> 1 /* Pozor na to !!! */
 SAY WORDS(TRANSLATE('Strom,kûñ,kañka',,',','))    -> 3
```

1.168 Popis funkce WRITECH()

```
                WRITECH(<logický soubor>,<êetëzec>)
```

Zapíše argument <êetëzec> do zadaného logického souboru.

Vrácená hodnota je poçet zapsaných znakû.

Na rozdíl od funkce

```
                WRITELN()
```

nepêídá znak pro posun êádku.

```
Pêíklad:
 SAY WRITECH('Výstup','Test')    -> 4
 SAY WRITECH(
                STDOUT
               ,'Já se pêítel.')
```

Význam posledního pêíkladu najdete u procedûry

```
                TISKNI()
               .
```

Podívejte se také na

```
                READLN()
```

a

```
                READCH()
               .
```

1.169 Popis funkce WRITELN()

```
                WRITELN(<logický soubor>,<êetëzec>)
```

Zapíše <êetëzec> do zadaného logického souboru a pêipojí znak pro posun êádkû.

Vrácená hodnota je poçet zapsaných êádkû.

Pokud nechcete pêipojit znak pro posun êádku, musíte pouùít podobnou funkci

```
                WRITECH()
               .
```

```
Pêíklad:
  SAY WRITELN('Výstup','Test')  -> 5
 Call WRITELN(
                STDOUT
               ,'Tohe je ùûùo')   /* Vypíše text do okna.  */
                                       /* Eqvivalentní s
                SAY
                 */

   Podívejte se také na
                READLN()
                a
                READCH()
               .
```

1.170 Popis funkce X2C()

```
                X2C(<êetëzec>)
```

Pêevede <êetëzec> sestávající z hexadecimálních çísel do (spakované) znakové podoby (podle ASCII tabulky).

Mezery jsou v argumentu <êetëzec> povoleny na hranicích bytu, jak je posáno v çásti zabívající se êetëzci .

```
 Opakem této funkce je
                C2X()
               .

Pêíklad:
 SAY X2C('12ab')                              -> "
 SAY X2C('12 ab')                             -> "
 SAY X2C(61)                                  -> a
 SAY X2C('41 68 6F 6A 20 6C C1 73 6B 6F')   -> Ahoj lásko

Podívejte se také na
                B2C()

                C2B()

                D2X()

                X2D()

                C2D()

                D2C()
```

1.171 Popis funkce X2D()

```
                X2D(<hexa êetëzec>[,<délka>])
```

Tato funkce pêemëní hexadecimální êetëzec na decimální çíslo.

V zápisu hexadecimálního êetëzce je moùno pouùívat mezer, tak jak je posáno v çásti zabývyjící se
```
                êetëzci
               }.
```

Opakem je funkce
```
                D2X()
               .
```

```
Pêíklad:
 SAY X2D('1f')           -> 31
 SAY X2D('7CF')          -> 1999
 SAY X2D('AB CD')        -> 43981

Podívejte se také na
                B2C()

                C2B()

                C2D()

                D2C()

                C2X()

                X2C()
```

1.172 Popis funkce XRANGE()

```
                XRANGE([<start>][,<konec>])
```

Funkce generuje êetëzec, který je tvoêen ze vîech znakû ASCII tabulky, jejichù kódy numericky leùí mezi zadanou startovní a koncovou pozicí. Pokud funkci zadáte bez parametû, bude vrácena celá ASCII tabulka. Je to jako kdyby bylo zadáno XRANGE('00'x,'FF'x).

Pokud zadáte argumenty <start> nebo <konec> delîí neù jeden znak, bude stejnë brán v úvahu jen jeden znak.

Jako parametry <start> a <konec> se udávájí znaky, nikoliv binarní nebo hexadecimální êetëzce. Pokud tedy chcete zadat napêíklad hexadecimální çíslo, musíte zadat: SAY XRANGE('41'X,'43'X). Pokud by jste ale chtëli pouùít desítkové çíslo, musíte jej nejprve pêevés na znak, k tomuto úçelu slouùí funkce
```
                D2C()
               .
```
Zápis by pak vypadal takto:
```
                                SAY XRANGE(D2C(65),D2C(68)).
```

```
Pêíklad:
```

```
 SAY XRANGE()                  ->  !"#$%&'()*+,-./0123456789:;<=>?@A .... atd.
 SAY XRANGE('a','f')           -> abcdef
 SAY
                C2X
               (XRANGE(,'0A'x))    -> 000102030405060708090A
 /* '0A'x znak pro odêádkování */
```

1.173 Vyzkouîejte si své znalosti v praxi.

5.5.3 Ukázkový program & Seznamte se s funkcemi

Následující pêíklad ilustruje mnohé integrované funkce pro manipulaci s êetëzci.

Program 13. Changestrings.rexx Spusï program !

```
                ! Poznámka !
               /*Tento   arexxový   program  ukazuje  pûsobení  integrovaných  ↩
                  funkcí  pro
manipulaci s êetëzci. Funkce se dají rozdëlit do dvou skupin: takové, které
mëní jednotlivé znaky a takové, které mëní celé êetëzce.*/

teststring1 = " every  good  boy  does  fine "

/*První skupinu tvoêí funkce STRIP(), COMPRESS(), SPACE(), TRIM(),
 TRANSLATE(), DELSTR(), DELWORD(), INSERT(), OVERLAY() a REVERSE(). */
/*
                STRIP()
                odstraní mezery z poçátku a konce*/
/*Vytisknëte  na  porovnání  originální êetëzec. êetëzec je uzavêen teçkou,
aby jste vidëli, co se stalo s mezerami.*/

SAY " every  good  boy  does  fine "

/* Ten samý êetëzec bez poçáteçních mezer */

SAY STRIP(" every  good  boy  does  fine ")"."

/*Pokus o odstranëní é-çek z konce a poçátku êetëzce*/

SAY STRIP(" every  good  boy  does  fine ",,"e")"."

/* "e" byly chránëny poçáteçními a koncovými mezerami. Pêi odstranëní mezer
budou "e"çka vystaveny pûsobení funkce STRIP(). */

SAY STRIP("every  good  boy  does  fine",,"e")"."

/*Nyní jsou "e"çka a mezery vymazány z originálního êetëzce */

SAY STRIP("every  good  boy  does  fine",," e")"."

/*Nyní  je nahoêe definovaná promënná "teststring1" pouùita. Odstranëní jen
koncových mezer z êetëzce*/

SAY STRIP(teststring1, T)"."
```

```
/*Odstranëní koncových mezer a "e" */

SAY STRIP(teststring1, T," e")"."

/*
                COMPRESS()
                odstraní vîechny  znaky z êetëzce. Zde jsou vîechny mezery z
êetëzce teststring1 smazány.*/

SAY COMPRESS(teststring1)

/* Dalîí çást se zabývá funkci
                TIME()
               . */

CALL TIME('r')
SAY TIME('Civil')    /*ças HH:MM (AM | PM) */
SAY TIME('h')        /*Hodiny od pûlnoci*/
SAY TIME('m')        /*Minuty od pûlnoci*/
SAY TIME('s')        /*Sekundy od pûlnoci*/
SAY TIME('e')        /*Uplynulý ças od spuîtëní çasového registru*/

/*Funkce:
                TRACE()
                Pouùití: TRACE([Volba]*/

SAY TRACE()
SAY TRACE(TRACE())   /*Ponechat nezmënëné */

/*Funkce:
                TRANSLATE()
                Pouùití: TRANSLATE(êetëzec[,Výstup][,Vstup][,Vyplñující
znak])*/

SAY TRANSLATE('aBCdef')
SAY TRANSLATE('abcdef','1234')
SAY TRANSLATE('654321','abcdef','123456')
SAY TRANSLATE('abcdef','123','abcdef','+')

/*Funkce:
                TRIM()
                Pouùití: TRIM(êetëzec)*/

SAY TRIM('    abc  ')

/*Funkce: TRUNC  Pouùití: TRUNC(çíslo[,poçet míst])*/
SAY TRUNC(123.456)
SAY TRUNC(134566.123,2) 'DM'

/*Funkce: UPPER  Pouùití: UPPER(êetëzec)*/
SAY UPPER('AbBCDef12Éb24')

/* Funkce: VALUE  Pouùití: VALUE(Jméno)*/
abc = 'Mé jméno'
SAY VALUE('abc')
```

```
/*Funkce: VERIFY  Pouùití: VERIFY(êetëzec,Seznam,[,'M'])*/
SAY VERIFY('123a45','0123456789')
SAY VERIFY('abc3de','012456789','M')

/*Funkce: WORD  Pouùití: WORD(êetëzec,n)*/
SAY WORD('Nyní je ças',3)

/*Funkce: WORDINDEX  Pouùití: WORDINDEX(êetëzec,n)*/
SAY WORDINDEX('Nyní je ças  ',3)

/*Funkce: WORDLENGTH  Pouùití: WORDLENGTH(êetëzec,n)*/
SAY WORDLENGTH('Nyní je jiù ças',4)

/*Funkce: WORDS  Pouùití: WORDS(êetëzec)*/
SAY WORDS('Nyní je jiù ças')

/*Funkce: WRITECH  Pouùití: WRITECH(Soubor,êetëzec)*/
IF OPEN('test','ram:test$$','W') THEN DO
    WRITECH('test','Hallo')   /*Zapsat êetëzec */
    CALL CLOSE 'test'
   END


/*Funkce: WRITELN  Pouùití: WRITELN(Soubor,êetëzec)*/
IF OPEN('test','ram:test$$','W') THEN DO
    WRITELN('test','Hallo')   /*Zapsat êetëzec + znak pro posun êádkû*/
    CALL CLOSE 'test'
   END

/*Funkce: X2C  Pouùití: X2C(hexadecimální êetëzec)*/

SAY X2C('616263')   /*Pêemëní na znaky(spakované)*/

/*Funkce: XRANGE  Pouùití: XRANGE([Start][,Konec])*/

SAY C2X(XRANGE('f0'x))
SAY XRANGE('a','g')
EXIT
```

Takto by vypadal výpis programu:

```
 every  good  boy  does  fine
every  good  boy  does  fine.
 every  good  boy  does  fine .
very  good  boy  does  fin.
very  good  boy  does  fin.
 every  good  boy  does  fine.
 every  good  boy  does  fin.
everygoodboydoesfine
3:55PM
15
955
57327
0.00
N
N
ABCDEF
```

```
abcdef
fedcba
123+++
    abc
123
134566.12 DM
ABBCDEF12éB24
Mé jméno
4
0
ças
9
3
4
abc
F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF
abcdefg
```

1.174 Pêíkazy knihovny RexxSupport.library

```
                5.6  Funkce knihovny REXXSupport.library
-----------------------------------------

V následující çásti  jsou výjmenovány funkce jenù  jsou souçástí  knihovny
REXXSupport.library.  Mohou  být  pouùity  jen  kdyù  je knihovna otevêená.
Následující pêíklad ukazuje, jak mûùete otevêít knihovnu.

 Program 14.  OpenLibrary.rexx        Spusï program !
                 ! Poznámka !
                /* Otevêít Rexxsupport.library, jestli se tak jiù nestalo */
IF ~ SHOW('L',"rexxsupport.library") THEN DO
/* Je uù otevêená? Není-li tedy, pokus se ji otevêít */
IF ADDLIB('rexxsupport.library',0,-30,0)
   THEN SAY "Rexxsupport.library otevêena."
  ELSE DO SAY 'Rexxsupport.library není k dispozici, program je opuîtën'
    EXIT 10  /* Pêi selhání ADDLIB je program opuîtën */
    END
 END

   Nemusíte  samozêejmë  pokaùdé  psát tak sáhodlouhý  program. Staçí, kdyù
napíîete:

IF ~SHOW('L','rexxsupport.library') THEN ADDLIB('rexxsupport.library',0,-30,0)

   Knihovna vîak musí existovat ve správném adresáêi !!!

 Poznámka: Pokud  budete  nëkdy  potêebovat pêipjit více knihoven a pokud
           budete chtít navíc otestovat zda kaùdá existuje je pohodlnëjîí
           napsat si k tomuto uçelu funkci.
```

1.175 Popis funkce ALLOCMEM()

Funkce je souçástí ↩
RexxSupport.library

```
 ALLOCMEM(<délka>[,<atribut>])
```

Rezervuje pamëïový blok dané délky z poolu volné systémové pamëti a vrátí tuto adresu ve formë 4-bytového êetëzce. Volitelný parametr <atribut> musí být standardní EXEC-ové oznaçení, které je zadáno jako 4-bytový êetëzec. Implicitní atribut je "MEMF_PUBLIC" pro vîeobecnë pêístupnou pamëï. Dalîí informace o typech pamëti a atributech najdete v popisu knihoven Amiga ROM Kernel Reference Manual (oficelní systémová dokumentace pro programátory). Tyto funkce by jste mëli pouùívat vùdy tam, kdyù má být rezervovaná pamëï pro externí programy.

Uvolnëní nepotêebné pamëti závisí na morálce uùivatele. Po skonçení programu vîak tato pamëï nebude vrácena zpët a hrozí zde i nebezpeçí zhroucení systému, proto je lepîí pamët opëï uvolnit pêíkazem

```
FREEMEM()
.
```

Pêíklad:

```
SAY C2X(ALLOCMEM(1000))    -> 00050000
SAY C2X(ALLOCMEM(1000,'00 01 00 01'x))  -> 00228400
/*1000 bytû vyCLEARované pamëti (nastavené na 0) vîeobecnë pêístupné pamëti*/
```

1.176 Popis funkce BADDR()

Funkce je souçástí RexxSupport.library

```
 BADDR(<BCPL adresový êetëzec>)
```

Tato funkce konvertuje adresu BPTR na CTPR adresu.
(pouze jesem to pêeloùil!)

1.177 Popis funkce CLOSEPORT()

Funkce je souçástí ↩
RexxSupport.library

```
 CLOSEPORT(<jméno>)
```

Uzavêe Message-port, který je zadán v argumentu <jméno>. Tento musí být pêedem uvnitê aktuálního arexxového programu voláním

```
OPENPORT()
otevêen.
```

Vîechny pêijmuté zprávy, na které jeîtë nebylo odpovëzeno (

```
REPLY()
```

), jsou
okamùitë vráceny s hodnotou 10.

Pêíklad:

```
CALL CLOSEPORT('MUJ_PORT')
```

S touto funkcí souvisí také
GETPKT()

GETARG()

WAITPKT()

1.178 Popis funkce DELAY()

Funkce je souçástí RexxSupport.library

DELAY(<číslo>)

Tato funkce pzastaví bëh programu. <číslo> udává prodlevu v 1/50 sekundy.

Bëhem çekání programu není zatëùován procesor, je tedy umoùnëno pohodlnëjîí zpracování dalîích taskû.

Pêíklad:

```
/* Çasová prodleva - TP */
Pocet = 5
SAY 'Çekám' pocet 'sekund'
Call Delay(pocet*50)       /* çeká -"pocet"- sekund */
SAY 'Hotovo!'
```

1.179 Popis funkce DELETE()

Funkce je souçástí RexxSupport.library

DELETE(<soubor>)

Tato funkce smaùe zadaný <soubor>.
Vratí booleovskou hodnotu udávající úspëînost.

Pêíklad:

```
Say Delete('ram:pokus.txt')     -> 0   /* tento soubor neexistuje */
```

1.180 Popis funkce FORBID()

Funkce je souçástí ↩
RexxSupport.library

FORBID()

Tato funkce slouùí k podchycení taskû napêíklad bëhem vaîeho pêístupu do pamëti. Pouùívá se pêedevîím pêi çtení promënných systémových seznamû. Pokud je totiù multitasking povolen, jiné tasky mûùou zpûsobit zmënu v seznamu, právë v okamùiku kdy je naçítán ARexxem, coù se nemûùe stát, pokud puùijete funkci FORBID(), protoùe ta pozastaví vîechny I/O operace.

V Amiga ROM kernel manuálu mûùete nalézt varování, ùe pêístup do pamëti bez zastavení multitaskingu mûùe poîkodit integritu celého systému. Interpet ARexxu pêi pêístupu do pamëti napê. funkcí

```
SHOWLIST()
```

zastavuje multitasking automaticky.

Funkce vratí çíslo oznaçující kolikrát byl jiù zákaz vydán (a nezruîen). Budete-li tuto funkci volat poprvé, bude návratová hodnota 0, dále pak 1,2,3, atd. Nevím vîak, jaký význam má funkce do sebe takto vnoêovat.

Pêed ukonçením programu není nutné multitasking opët zapínat (po ukonçení programu se tak stane automaticky), ale je vùdy lepîí, kdyù takto uçiníte co nejdêíve...

Opakem této funkce je funkce

```
PERMIT()
```

, která multitasking opët povolí.

Pêíklad pouùití této funkce si mûùete prohlídnout v programu "

```
Adresy
```

"

1.181 Popis funkce FREEMEM()

Funkce je souçástí ↩ RexxSupport.library

```
FREEMEM(<adresa>,<délka>)
```

Uvolní pamëïový blok dané délky (pêedá na vnitêní pool systémové pamëti. Argument <adresa> je 4-bytový êetëzec, který obvykle vrátí funkce

```
ALLOCMEM()
```

. FREEMEM() nemûùe být pouùito k uvolnëní pamëti, rezervované interní arexxovou funkcí

```
GETSPACE()
```

.

Vrácená je

booleovská hodnota udávající úspëch (1) nebo selhání (0).

Pêíklad:

```
MEM_SIZE = 1024                          /* definice velikosti pamëti*/
Adr = ALLOCMEM(Mem_Size)                 /* alokování pamëti */
SAY  C2X(Adr)               -> 07C987B0  /* moùná adresa rezervovaného bloku */
SAY FREEMEM(Adr,Mem_Size)   -> 1         /* uvolnëní pamëti */
```

Pozor !!!

Pêed ukonçením programu je vhodné (spíî nutné) funkcí FREEMEM() uvolnit pamëï, která byla pêedtím rezervována funkcí

```
ALLOCMEM()
```

. Pamët totiù po ukonçení programu není vrácena zpët. Byla by vám v takovém pêípadë

k dispozici aù po restartu poçítaçe. Navíc hrozí i nebezpeçí zhroucení systému !!

1.182 Popis funkce GETARG()

Funkce je souçástí ↩
RexxSupport.library

GETARG(<paket>[,<pozice>])

Extrahuje (vyêízne) kommando, funkçní jméno nebo argumentový êetëzec ze zprávového balíku (message packet). Argument <paket> musí být 4-bytová adresa, která byla získána z pêedchozího volání funkce

GETPKT()

.

Volitelný argument <pozice> oznaçuje pozici, kde stojí êetëzec k extrahování. <pozice> musí být menîí nebo rovna poçtu argumentû v paketu. Pêíkazy a funkçní jména se nachází na pozici 0. Funkçní balíky mohou vykazovat argumentové êetëzce na pozicích 1 aù 15.

Pêíklad:
Kommando = GETARG(Paket)
Funkce = GETARG(Paket,0)
Arg1 = GETARG(Paket,1)

S touto funkcí souvisí také

OPENPORT()

CLOSEPORT()

REPLY()

GETPKT()

WAITPKT()

1.183 Popis funkce GETPKT()

Funkce je souçástí ↩
RexxSupport.library

GETPKT(<jméno>)

Pêezkouîí, zda na message-port <jméno> pêiîly nëjaké zprávy. Zadaný Message-port musí být pêedem otevêen uvnitê aktuálního arexxového programu voláním

OPENPORT()

.

Vrácená hodnota je 4-bytová adresa prvního zprávového paketu nebo '0000 0000'x (tento êetëzce vrací funkce

NULL()
), kdyù zde ùádné nejsou.

Programy by nemëly být nikdy uspoêádány tak, aby v úzké smyçce volaly permanentnë tuto funkci. Naopak, pokud program nemá aù do pêíchodu dalîího zprávového paketu co na práci, mëlá by se pouùít funkce
WAITPKT()
, která
umoùní zpracování ostatních taskû.

Pêíklad:
packet = GETPKT('MUJ_PORT')

S touto funkcí souvisí také
CLOSEPORT()

REPLY()

GETARG()

1.184 Popis funkce NEXT()

Funkce je souçástí RexxSupport.library

NEXT(<adresa>[,<offset>])

Returns the 4-byte address string at <address> plus <offset>. The function combines features of import() and offset() . Like import(), it reads a value from memory, but is designed for the specific task obtaining an address. Like offset(), it will, when given a decimal offset, calculate a new address in the proper format.

A linked-list maintained by the operating system can be followed by using the following format:

NextNode = NEXT(<node-address>)
PrevNode = NEXT(<node-address>, 4).

See example at IMPORT()

Also see OFFSET

The base address of most system resources can be obtained with the SHOWLIST() function, using its fourth 'Address' argument.

1.185 Popis funkce NULL()

Funkce je souçástí ↩
RexxSupport.library

NULL()

Tato funkce vrátí "null pointer" jako 4 bytový êezezec ('0000 0000'x).

```
  Napêíklad funkce
                GETPKT()
                vrátí  tento  êetëzec pokud na daný meessage
port nepêíjde  ùádná  zpráva, takùe pokud chcete testovat zda pêiîla nëjako
správa staçí napsat:

  /* Test nové zprávy */
  ...
  p = GetPkt('POKUS_PORT')

                IF
                p~== NULL() Then Do
    arg =
                GetArg
               (p)
    ...
  End
  ...

  Podívejte se také na funkcí
                OFFSET()
```

1.186 Popis funkce OFFSET()

```
                                                        Funkce je souçástí  ↩
                                                           RexxSupport.library
 OFFSET(<adresa>,<posun>)

   Funkce  vrací  adresu,  která  je  od  vstupní  adresy <adresa> posunuta
o parametr  <offset>.  Vstupní  <adresa>  musí  být  zadána  jako 4-bytový
êetëzec.  Stejný  formát  má také výsledek funkce.  <offset> musí bý zadán
jako celé çíslo v desítkovém tvaru (ne hexadecimálním).

   Výhodou této funkce je, ùe nemusíte pouùívat funkcí
                C2D()
                a
                D2C()
               .

 Pêíklad:
  adr = '0000 0000'x
  SAY c2x(offset(adr,4))      ->  00000004
  SAY c2x(offset(adr,-4))     ->  FFFFFFFC

   Podívejte se také na funkci
                NEXT()
               .
```

1.187 Popis funkce OPENPORT()

Funkce je souçástí ↩
RexxSupport.library

OPENPORT(<jméno>)

Vytvoêí veêejnë pêístupný "Message port" se zadaným jménem. Vrácená booleovská hodnota udává úspëîné nebo neúspëîné otevêení portu. Kdyù jiù port se zadaným jménem existuje nebo signální bit nemohl být rezervován, dojde k inicializaçní chybë.

Message-port je zaloùen jako "port-ressource-uzel" a integrován do globální datové struktury. Porty jsou automaticky uzavêeny pêi opouîtëní programu a nezpracované zprávy poslány zpët odesílateli.

Port lze opët zavêít funkcí
CLOSEPORT()
.

Pêíklad:
```
SAY OPENPORT("MUJ_PORT")
```

S touto funkcí souvisí také
REPLY()

GETPKT()

GETARG()

WAITPKT()

1.188 Popis funkce PERMIT()

Funkce je souçástí ↩
RexxSupport.library

PERMIT()

Tato funkce je opakem funkce
FORBID()
,opët tedy povoluje multitasking.

Návratovou hodnotou funkce je çíslo, které vrátila pêedtím funkce FORBID(), avîak je zmenîeno o 1. Voláte-li funkci znovu je vráveno çíslo opët o 1 menîí. Vrátí-li tedy funkce -1 (nebo menîí) máte jistotu, ùe multitasknig byl jiù povolen. Nevým vîak jaký význam má funkci FORBID() volat vícekrát za sebou ?

Pêíklad pouùití této funkce mûùete vidët u ukázkového programu
Adresy
.

1.189 Popis funkce REPLY()

Funkce je souçástí ↩
RexxSupport.library

```
REPLY(<paket>,<rc>)
```

Poîle zprávový balík zpët odesílateli. Primární výsledkové pole je nastaveno na hodnotu argumentu <rc>. Sekundární výsledek je smazán.

Argument <paket> má být zadán ve formë 4-bytové adresy, kterou vrácí funkce

GETPKT()

. A <rc> musí být zadáno jako celé çíslo.

Pêíklad:

```
CALL REPLY(Paket,10)    /*Vrácení chyby*/
```

1.190 Popis funkce SHOWDIR()

Funkce je souçástí ↩
RexxSupport.library

```
SHOWDIR(<adresáê>[,'ALL'|'FILE'|'DIR'][,<oddëlující znak>])
```

Vrátí obsah zadaného adresáêe jako posloupnost jmen, oddëlených od sebe mezerami nebo oddëlujícím znakem.

Druhý parametr je klíçové slovo volby, pêes který je êízeno, zda ma být vypsáno vîechno ('ALL'), jen soubory ('FILES') nebo jen podadresáêe ('DIR') dotyçného adresáêe. Implicitní volba je 'ALL'. U kaùdé volby staçí uvést pouze první písmeno.

Cestu lze zadat i relativnë vzhledem k adresáêi ze kterého byl program spuîtën nebo který byl nastaven funkcí

PRAGMA()

.

Pêíklad:

```
SAY SHOWDIR('Ram:',ALL,'*')  -> ENV*Clipboards*T*Disk.info / * Napêíklad */
SAY SHOWDIR('T:',DIR)        ->    /* Ùádné adresáêe tam nejsou. */
```

1.191 Popis funkce SHOWLIST()

Funkce je souçástí ↩
RexxSupport.library

```
SHOWLIST(<volba>[,<jméno>][,<oddëlující znak>])
```

Parametrem <volba> mûùete vybrat nëkterý z následujících seznamû:

```
Assigns and Assigned Devices  -   Pêiêazení a pêiêazená zaêízení.
Device Drivers                -   Ovladaçe zaêízení
```

```
 Handlers                    -   Vstupní/Výstupní programy
 Interrupts                  -   Pêeruîení
 Libraries                   -   Knihovny
 Memory List Items           -   Zápisy pamëïového seznamu
 Ports                       -   Porty
 Resources                   -   Ressource nebo provozní prostêedky
 Semaphores                  -   Semafory
 Tasks (Ready)               -   Pêipravené tasky
 Volume Names                -   Jména datových nosiçû
 Waiting Tasks               -   Stojící tasky
```

Nepouùijete-li parametr <jméno>, funkce pouze vrátí êetëzec v nëmù budou dané poloùky oddëleny mezerou nebo zadaným vyplñujícím znakem.

V pêípadë, ùe pouùijete argument <jméno>, hledá funkce výskyt jména v daném seznamu a vrátí

```
booleovskou hodnotu
informující o tom,zda poloùka
```

<jméno> v existuje nebo ne.

U jmen je rozliîováno mezi malými a velkými písmeny !

Pêi prohledávání seznamu jsou podchyceny tasky, aby vám byl k dispozici exaktní a aktuální seznam.

Poslední volitelný parametr 'Address' nebo 'A' funkci naêizuje, aby vracela êídící adresu poloùky specifikované argumentem <jméno>. Lze tak zjistit, kde se v pamëti nachází velice zajímave vëci. K tomu aby jste toho mohly nëjak vyuùít musíte znát systém Amigy. Mûùet prostudovat napêíklad "Amiga Rom kernel manuál". Já osobnë jej také némám,ale mûùu vám ukázat jeden pêíklad. Podívejte se na ukázkový program "

```
Adresy.rexx
```

".

Pêíklad:

```
SAY SHOWLIST('P')         ->  ConClip.rendezvous REXX ARexx ProWrite
SAY SHOWLIST('P',,';')    ->  ConClip.rendezvous;REXX;ARexx;ProWrite
SAY SHOWLIST('P','REXX')  -> 1
SAY
```

```
C2X
(showlist(L,'amigaguide.library',,A)) -> 6820D5B4 /* napêíklad! */
```

Pokud nebudete vyùadovat nëkteré vymoùenosti této funkce, mûùete pouùít jednoduîîí integrovanou funkci

```
SHOW()
```

.

1.192 Popis funkce STATEF()

Funkce je souçástí ↩
RexxSupport.library

```
STATEF(<název souboru>)
```

Vrátí êetëzec s informací o externím souboru.

Êetëzec má následující formát:

```
 [DIR | FILE] Délka Bloky Ochrana Dny Minuty Tiky Komentáê

DIR|FILE - informuje o tom, zda se jedná o soubor nebo adresáê
Délka    - oznaçuje délku souboru v bytech.
Blok     - délku v blocích.
Ochrana  - standartní zpûsob informace o stavových bitech souboru.

Následující údaje se vztahují k datumu poslední zmëny v programu:
Dny      - poçet dní od 1.1.1978
Minuty   - poçet minut od pûlnoci
Tiky     - poçet "tikû" (1/50 sekundy) v dané minutë

Komentáê - komentáê k souboru

Poznámka: Prostudujte  "Pêíruçku uùivatele II", protoùe  vîechny tyto údaje
          jsou souçásti informací o ikonë.

Pêíklad:

SAY STATEF("LIBS:REXXSupport.library")
/* Moùný výsledek */  -> FILE 2524 5 ----RW-D 5468 910 2208
```

K rozkladu tohoto êetëzce lze pouùít syntaxní analýzu, která je êízena pêíkazem PARSE.

1.193 Popis funkce TYPEPKT()

Funkce je souçástí ↩ RexxSupport.library

```
 TYPEPKT(<adresa>[,<mód>])
```

Tato funkce slouùí ke zjiîtëní infomací o zprávovém balíku (message packet) na adrese <adresa>. Adresa musí mít formát 4-bytového êetëzce, který se získá funkcí GETPKT().

Pokud nepouùijete parametr <mód>, vrátí funkce zabalený 4-bytový êetëzec z informacemi. Pokud chcete z tohoto êetëzce informace pêímo získat, mûùete jako parametr <mód> pouùít nëkteré z následujících klíçových slov: (staçí pouùíz první písmeno)

Mód	Popis vrácené hodnoty a její pozice v êetëzci
Arguments	Vrátí návratovou hodnotu argumentû. Tato infomace je obsaùena v nultém bytu nepakovaného êetëzce

Command Vrátí 1 pokud byl paket volán jako pêíkaz. Tato informace je obsaùena v 3 bytu vráceného êetëzce, jehoù hodnot pak je '01'x

Function Vrátí 1 pokud byl paket volán jako funkce. Tato informace je obsaùena v 3 bytu vráceného êetëzce, jehoù hodnota pak je '02'x

Druhý byte zabeleného êetëzce scpecifikuje modifikaci vlajek, které byly nastaveny, kdyù byl paket volán. Funkce REPLY() automaticky manipuluje s nëkterou z modifikací nastavenou voláním pêíkazu nebo funkce.

Pêedcházející text jsem pouze pêeloùil (jestli se tomu tak dá êíkat) a jelikoù jsem zcela nepochopil k çemu vlastnë tahle funkce slouùí a navíc pêi mých pokusech se mi nepodaêilo z funkce získat jiný êetëzec neù tento: '01020000'. Takùe si teð nejsem ani jistý jestli je pêedcházející pêeklad vûbec pravdivý a zda jsou ta správná çísla na správných bytech?

V anglickém originále bylo jeîtë napsáno, ùe i pêesto ùe tato funkce vypadá zbyteçná, je "uùiteçná v prototypech arexxového interfejsu pêi pêenosu do niùîích jazykû" ???

1.194 Popis funkce WAITPKT()

Funkce je souçástí ↩
RexxSupport.library

WAITPKT(<jméno>)

Çeká na Message-portu <jméno> na pêijmutí zprávy. Tento port musí být otevêen uvnitê arexxového programu voláním funkce
OPENPORT()
.

Booleovská
vrácená hodnota urçuje, jestli na portu leùí zprávový paket. Normálnë je vrácená hodnota 1, protoùe funkce çeká na portu tak dlouho, dokud se nëco na daném portu nedëje.

Paket musí být následnë naçten funkcí
GETPKT()
a funkcí
REPLY()
opët poslán zpët. Vîechny pêijmuté, ale neodeslané zprávové ↩
pakety jsou
pêi opuîtëní programu automaticky zodpovëzeny s návratovou hodnotou 10.

Pêíklad:
```
CALL WAITPKT('MUJ_PORT')    /*çekat*/
```

S touto funkcí souvisí také
CLOSEPORT()

GETARG()

1.195 6. kapitola, Monitorování a Preruîení (Interrupts)

Hledání chyb, Monitorování a trasování

ARexx nabízí funkce na hledání chyb (debugging) a monitorování, trasování (tracing) na úrovni zdrojového textu.

Trasování ukazuje v prûbëhu programu vybrané pokyny uvnitê tohoto programu. Narazí-li trasovací proces na klausuli, je potom çíslo jejího êádku, zdrojový text a informace, týkající se jí, zobrazeny na konzoli.

Monitorování
Interní systém pêeruîení dává arexxovému programu moùnost ↩
, urçité
synchronní nebo asynchronní události rozpoznat, a odpovídajícím zpûsobem na në reagovat. Události jako syntaxní chyby nebo externí poùadavky na pêeruîení, které by normálnë vedly k opuîtëní programu, mohou být tímto zpûsobem odchyceny a vhodným opatêením oîetêeny.

Pêeruîení (Interrupts)

1.196 Monitorování

6.1 Monitorování

Monitorování vám nabízí velice výkonný nástroj pêi kontrole a odlaðování vaîeho programu. Máte moùnost si vybrat z nëkolika typû monitorování, dále mûùete sami êídit výstup monitorování a dokonce do bëùícího programu interaktivnë zasahovat a ovlivòovat jeho çinnost.

Máte k dispozici 10 monitorovacích voleb. Monitorovací volba urçuje, které klausule zdrojového textu mají být sledovány, dále vlastní dvë modifikaçní znaçky k êízení
kommandové záchytky
a trasování v
interaktivním reùimu
.
Monitorovací volby mohou být zadány jediným písmenem.

Moùné volby jsou:

ALL Vîechny klausule spadají pod trasování.

BACKGROUND Nekoná se monitorování a program nemûùe být donucen k interaktivnímu monitorování.

COMMANDS Vîechny kommandové klausule spadají pod monitorování, neù jsou poslány na externí host. Vrácené hodnoty rûzné od

nuly jsou poslány na konzoli.

ERRORS Kommanda, která generují hodnotu rûznou od nuly, jsou sledována po provedení klausule.

INTERMEDIATES Vîechny klausule spadají pod monitorování. Bëhem vyhodnocování výsledkû jsou zobrazovány dílçí výsledky. K tëmto výsledkûm patêí hodnoty vyvolané z proměnných, rozîíêená sloùená jména a výsledky volání funkcí.

LABELS Vîechny klausulové skokové znaçky jsou sledovány bëhem jejich provádëní. Skoková znaçka je vùdy tehdy ukázána, kdyù je êízení pêeneseno na jiné místo.

NORMAL Kommandové klausule, jejichù vrácená hodnota pêekraçuje (standardní) aktuální chybové hranice,jsou po provedení vzaty na monitorovací seznam a je zobrazeno chybové hláîení.

OFF Monitorování je vypnuto.

RESULTS Vîechny klausule jsou pêed provedením zapsány do monitorovacího seznamu, a výsledky kaùdého jednotlivého výrazu jsou zobrazeny. Hodnoty, které byly proměnným pêiêazeny pêíkazem ARG, PARSE nebo PULL, jsou rovnëù zobrazeny. Tato volba je doporuçena, kdyù chcete program podrobit dûkladnému testu.

SCAN Toto je speciální funkce, která vîechny klausule zapíše do monitorovacího seznamu a prohledá na chyby, ale zabrání vykonání pokynû. Hodí se obzvláîtë na první kontrolu novë napsaných programû.

Monitorovací mód mûùe být nastaven pêíkazem

TRACE

nebo integrovanou

funkcí

TRACE()

. Monitorování mûùe být bëhem programu selektivnë inaktivováno k pêeskoçení uù testovaných çástí programu.

Kaùdý êádek monitorování, zobrazený na konzoli, je posunut, aby bylo zvýraznëno skuteçné êízení popê. vnoêení do sebe. Dále je kaùdý êeádek oznaçen specielním trojmístným kódem (viz tabulka). Zdrojovému textu kaùdé klausule je çíslo êádku bëhem programu pêedêazeno. Výsledky a mezivýsledky výrazû stojí v dvojitých uvozovacích znacích, aby byly zêetelné i mezery. Speciální têímístné kódy.

Kód	Zobrazená hodnota
+++	Kommandová nebo syntaxní chyba
>C>	Rozîíêené sloùené jméno
>F>	Výsledek volání funkce
>L>	Skoková klausule
>O>	Výsledek dyadické operace
>P>	Výsledek prefixové operace

```
      >U>     Neinicializovaná promënná
      >V>     Hodnota promënné
      >>>     Výsledek výrazu nebo îablony
      >.>     Hodnota místodrùícího tokenu
    ------------------------------------------------------------------------

   ARexx umoùñuje
                výstupní data monitorování
                posílat na jiný výstup  neù
standartní výstup programu (
                STDOUT
               ).

arexxové monitorování dále umoùñuje:

                   Oîetêení chyb pêi Interaktivním reùimu

                Zvenku donutit bëùící program k monitorování
               Pokud jste jiù  dêíve neprostudovali  dva speciální reùimy, je ↩
                  nejvyîîí ças
takto uçinit:
                Kommandová záchytka

                Interaktivní monitorování
```

1.197 Výstupní data monitorování

```
                6.1.1  Výstupní data monitorování

Výstupními daty pêi monitorování programu jsou vùdy nakrmeny jeden nebo dva
datové proudy.  Interpreter hledá nejprve datový proud se jménem
                STDERR
                a
vede  výstup  tam,  pokud  tento datový proud existuje. Jinak jde výstup na
standardní  výstupový  proud
                STDOUT
                a  je  smíchán s normálním konzolovým
výstupem  programu. Datové  proudy  STDERR  a  STDOUT mohou být otevêeny a
uzavêeny  programem,  takùe programátor má stále cíl monitorovacího výstupu
pod kontrolou.

V  mnohých  pêípadech  není  výstupový datový proud pêedem definován. Je-li
napêíklad  program  zavolán  host-aplikací,  která  nepêipravila
vstupní/výstupní datové proudy, nedisponuje potom program výstupní konzolu.
Aby  byla pro takové programy umoùnëno monitorování, mûùe residentní proces
otevêít speciální, globální monitorovací konzolu, která mûùe být napíchnuta
kaùdým  aktivním programem. Je-li tato konzola otevêena, otevêe interpret
automaticky  datový  proud  se jménem STDERR pro kaùdý program, kterém není
STDERR  definován.  Program  pêevede svûj monitorovací výstup na tento nový
datový proud.

S pomocným  kommadovým programem
                TCO
                (TraCe  Open) mûùe být konzola pro
```

globální monitorování otevêena. Arexxové programy pêevedou výstupní data jejich monitorování automaticky do nového okna, které je otevêeno jako standardní AmigaDOSová konzola. Uùivatel mûùe libovolnë zmënit pozici a velikost tohoto okna.

Monitorovací konzola slouùí také jako vstupní datový proud pro programy bëhem
interaktivního monitorování
. Kdyù program çeká na vstupní data k
monitorování, musí tato data být zadána pêes monitorovací konzolu (okno). Libovolnë mnoho programû se mûùe dëlit o monitorovací konzolu. Doporuçuje se ale aktivovat monitorování vùdy jen pro jeden program. Globální konzola mûùe být zavêena kommandem
TCC
(TraCe Close). Avîak uzavêení je
provedeno jen tehdy, kdyù vîechny poùadavky na çtení z konzoly byly zpracovány. Konsola je také zavêena jen tehdy, kdyù vîechny ostatní programy nahlásí, ùe ji uù nepotêebují.

1.198 Kommandová záchytka

6.1.2 Kommandová záchytka

ARexx disponuje monitorovacím módem, který potlaçuje host-kommanda, tzv. kommandová záchytka. V tomto módu jsou kommandové klausule, jak je zvykem, vyhodnoceny, ale kommando není ve skuteçnosti posláno na externí host. Vrácená hodnota (RC) je nastavena na nulu. Tímto zpûsobem mohou být testovány programy, které vydávají kommanda s potencielnë niçivým úçinkem napê. mazání souborû nebo formátování disku nebo "pêeùvykování" disket. Kommandová záchytka neplatí ale pro klausule, které byly zadány v dialogovém reùimu (interaktivnë). Taková kommanda jsou vùdy provedena, ale hodnota zvláîtní promënné RC zûstane nezmënëna.

Kommandová záchytka mûùe být nasazena ve spojení s kaùdou monitorovací volbou. Je êízena znakem "!", osamëlým nebo pêed abecední volbou v pêíkazu

TRACE
. S kaùdým "!" je momentálnë aktivní záchytkový mód pêepnut do ↩
druhého
módu. Kommandová záchytka je zruîena, kdyù je monitorování nastaveno na nulu.

1.199 Interaktivní monitorování

Monitorování - popis
6.1.3 Interaktivní monitorování

Interaktivní monitorování je pomûcka pêi hledání chyb, která umoùñuje uùivateli bëhem provádëní programu zadávat zdrojové pokyny. Pêes takovéto pokyny mohou být hodnoty promënných pêezkouîeny nebo zmëněny, vydány

kommanda, nebo nëjakým zpûsobem vstoupit do dialogu s programem. Kaùdý platný pokyn mûùe být interaktivnë zadán. Platí stejná pravidla a omezení jako pro pêíkaz

INTERPRET

. Obzvláîtë je têeba dávat pozor, ùe sloùené pokyny jako napê.

DO
nebo
SELECT
musí být uvnitê zadané êádky kompletní.

Poznámka: Jednotlivé pokyny se samozêejmë oddëlují stêedníkem ";".

Interaktivní monitorování mûùe být pouùito s jakoukoliv monitorovací volbou. V dialogovém módu vloùí interpret pauzu po kaùdé klausuli, zapsané v monitorovacím seznamu, a vyùádá si kódem ">+>" uùivatelský vstup.

Pêi kaùdé takovéto pauze má uùivatel na výbër z têí rûzných odpovëdí:

* Zadání nulového êádku (nic) - program pokraçuje k dalîí pauze.

* Zadání "=" - pêedchozí klausule je provedena jeîtë jednou.

* Kaùdý dalîí vstup je jako arexxový pokyn; jako takový je pêezkouîen a vyhodnocen.

Protoùe se interpret po povedeném monitorování klausule pauzne, jsou tyto pauzy êízeny aktivními monitorovacími volbami. Provedení nëkterých pêíkazû nemûùe být opakováno bez rizika, ùe se interpret po provedení takových pêíkazû nepauzne. Tyto pêíkazy jsou

CALL
,
DO
,
ELSE
,
IF
, THEN

a

OTHERWISE
.

Interpreter se nezastaví po klausulích, které generovaly pêi provedení chybu.

Interaktivní monitorování je êízeno znakem "?", který stojí osamële nebo s abecední volbou v TRACE pêíkazu. Pêed volbou mûùe stát libovolnë mnoho "?". S kaùdým "?" je aktivní mód pêepnut do druhého módu. Byla-li napêíklad aktuální volba NORMAL, tak by byla nastavena pêíkazem "TRACE ?R", volba RESULTS a aktivováno interaktivní monitorování. Pozdëjîí "TRACE ?" by zruîil interaktivní monitorování.

1.200 Oîetêení chyb

6.1.3.1 Oîetêení chyb

Arexxový interpret nabízí speciální oîetêení chyb pêi dolaðování vaîeho programu.

Během interaktivního hledání chyb jsou nalezené chyby hláîeny, ale nevedou k ukonçení bëhu programu. Toto speciální zpracování platí ovîem jen pro pokyny zadané v dialogovém (
interaktivním
) reùimu.

ARexx inaktivuje také
interní pêeruîovací znamení
během interaktivního
hledání chyb. Tím je zabráněno nechtënému pêenosu êízení, které by mohlo být vyvoláno chybou nebo neinicializovanou promënnou. Je-li ale zadán pêíkaz "
SIGNAL
<Skoková_znaçka>" je pêenos êízení vykonán a na eventuelní pozûstalá interaktivní vstupní data není brán zêetel. Pêíkaz
SIGNAL
mûùe
být pouùit také k zmënë pêeruîení. Nová nastavení vstoupí v platnost, kdyù se interpret vrátí do pûvodního pracovního módu.

Kaùdý arexxový task inicializuje hraniçní hodnotu chyb vzhledem ke klientovi (bëùnë 10). Tím je podchycen výstup nepotêebných kommandových chyb. Hraniçní hodnota chyb mûùe být zmënëna
OPTIONS
FAILAT.
Mûùete tak sami urçit mez pêi které bude návratová chybová hodnota brána jeîtë za malou chybu a kdy za úplné selhání. Pokud bude návratová chybová hodnota (RC) menîí neù vámi nastavená mez (pêíkazem
OPTIONS
FAILAT), bude
êízení programu pêedáno na návëstí definované pêíkazem
SIGNAL
ON ERROR a
pokud bude vëtîí nebo rovna mezní hranici dojde k pêedání êízení na návëstí, jenù bylo definováno pêíkazem
SIGNAL
ON FAILURE.

1.201 Externí monitorovací poznávací znamení

6.1.3.2 Externí monitorovací poznávací znamení

ARexx disponuje externím monitorovacím poznávacím znamením, s kterým mohou být donuceny programy do interaktivního monitorování. Kdyù je toto znamení vloùeno (pomocným programem
TS
(Trace Set)), pêejde kaùdý program do

interaktivního monitorování
(pokud tam není). Interní volba monitorování
je nastavena na RESULTS, pokud není zrovna nastavena na INTERMEDIATES nebo

SCAN. V tëchto pêípadech zûstává nezmënëna. Programy, které jsou volány, kdyù je externí monitorovací poznávací znamení nastaveno, zaçínají provedení v módu interaktivního monitorování.

Externí poznávací znamení monitorování nabízí moùnost, získat kontrolu nad programy, které jsou zacykleny nebo nereagují. Jakmile pêejde program do interaktivního monitorování, mûùe uùivatel projít programové pokyny krok za krokem a diagnostikovat problém. Externí monitorování je globální poznávací znamení - pûsobí na vîechny aktivní procesy. Poznávací znamení monitorování zûstává tak dlouho nastaveno, dokud není smazáno pomocným kommandovým programem

TE

(Trace End). Kaùdý program si uschovává svou vlastní interní kopii posledního statusu poznávacího znamení monitorování a nastaví svoji vlastní monitorovací volbu na OFF, kdyù zjistí, ùe znamení monitorování bylo smazáno. Programy v monitorovacím módu BACKGROUND nereagují na externí monitorovací poznávací znamení !

1.202 Êízení Pêeruîení, aneb jak odstaranit chyby ?

6.2 Pêeruîení (Interrupts)

ARexx podporuje interní systém pêeruîení k poznání a odchycení nëkterých chybových podmínek. Je-li aktivováno pêeruîení a jeho odpovídající podmínka je platná, dojde k pêenosu êízení na pêeruîení naleùící skokovou znaçku. Tímto zpûsobem si mûùe program udrùet êízení, i kdyù by dané (pêedloùené) podmínky normálnë vedly k pêeruîení programu. Interruptové podmínky mohou být vyvolány synchronní událostí (syntaxní chyba) nebo asynchronní událostí (poùadavek na pêeruîení CTRL-C).

Taková interní pêeruîení nemají nic spoleçného s hardwarovým systémem pêeruîení, který je spravován provozním systémem EXECem.

Jméno pêiêazené interruptu je vlastnë skoková znaçka, na kterou je pêedáno êízení. Takùe interrupt SYNTAX pêenese êízení na skokovou znaçku "SYNTAX:". Interrupty mohou mohou být aktivovány nebo deaktivovány pêíkazem

SIGNAL

.

Pêíkaz SIGNAL ON SYNTAX aktivuje interrupt SYNTAX.

Interrupty podporované ARexxem jsou:

BREAK_C Zachytí AmigaDOSem generovanou ùádost o pêeruîení CTRL-C (to znamená, ùe je zaregistrována a jak o signál interpretována, ale ne jako normální vstup). Není-li interrrupt aktivován, program se okamùitë zastaví, vydá hláîení "Execution halted" a vrátí chybový kód 2.
Poznámka: Pokud chcete beùící program zastavit musíté poùadavek o zasatavení poslat pêímo programu. Pokud tedy jen tak nëkde stisknete CTRL+C,arexxový program se o tom vûbec nedozví. Musíte tyto klávesy stisknout pêi aktivovaném výstupním nebo vstupním oknë. Pokud program ùádný vstup nebo výstup nemá, mûùete jej buð zastavit pomoci programu

```
 HI
 nebo jej  s pomocí  programu
 TS
                donutit k
 interaktivnímu monitorování
.
```

BREAK_D Zachytí ùádost o pêeruîení generovanou AmigaDOSem CTRL-D. Není-li interrupt aktivován, je poùadavek o pêeruîení ignorován.

BREAK_E Zachytí ùádost o pêeruîení generovanou AmigaDOSem CTRL-E. Není-li interrupt aktivován, je poùadavek o pêeruîení ignorován.

BREAK_F Zachytí ùádost o pêeruîení generovanou AmigaDOSem CTRL-F. Není-li interrupt aktivován, je poùadavek o pêeruîení ignorován.

ERROR Tento interrupt je generován kaùdým host-kommandem, které vrátí chybovou hodnotu rûznou od nuly.

FAILURE Tento interrupt odchytí chyby jejichù chybová hodnota je vëtîí neù mez nastavená pêíkazem
OPTIONS
 FAILAT. Lze tak
docílit toho, aby byly malé chyby odchyceny interruptem ERROR a ty vëtîí tímto.

HALT Tímto interrupt je odchycen kaùdý externí poùadavek na pêeruîení. To mûùe být vydáno pomocným programem HI . Pokud není tento interrupt aktivní, vydá program toto hláîení "Execution halted".

IOERR Je-li aktivován, jsou potom chyby pêi I/O operacích (vstup/výstup) odchyceny.

NOVALUE Je-li aktivována tato podmínka, dojde k pêeruîení, kdyù je pouùita neinicializovaná promënná. Spuîtëní tohoto interruptu mûùe nastat napêíklad bëhem volání funkce

```
 UPPER()
 nebo integrované funkce
 VALUE()
.
```

SYNTAX Tento interrupt odchytí syntaxní chybu nebo chybu v provádëní. Ne vîechny chyby tohoto druhu mohou být odchyceny.Nëkteré se vyskytnou pêed provedením programu. Takové,které byly odhaleny externím arexxovým rozhraním, nemohou být interruptem SYNTAX odhaleny. Odchytit nelze tyto chyby: 1, 3 aù 8, 10, 11. Ty jsou vîak ohláîeny jeîtë pêed spuîtëním samotného programu, takùe se nemûùe stát, ùe se projeví aù nëkdy pozdëji, jako tomu mûùe být u ostatních chyb.

Kdyù interrupt násilnë pêivodí pêenos êízení, jsou k tomuto okamùiku aktivní êídící oblasti inaktivovány. Interrupt, který vyvolá pêenos êízení, je inaktivován. To zabrañuje vzniku rekursivní interrupt smyçky. Týká se ale jen êídících struktur aktuálního prostêedí. Interrupt generovaný uvnitê funkce nemá vliv na úroveñ, z které byla volána.

Z toho vyplýva, ùe pokud dojde k chybë uvnitê procedûry (interní funkce) a procedûra pro oîetêení chyby je ukonçena pêíkazem

```
RETURN
```

, je êízení vráceno hned za volání procedûry, která prve zpûsobila chybu.

Výskyt pêeruîení se odráùí na dvou zvláîtních proměnných:

SIGL Je vùdy nastaveno na çíslo aktuálního êádku, dokud nedojde k pêenosu êízení. Tím mûùe být zjiîtëno, která êádka zdrojového textu byla prováděna.

RC Je nastaven na chybový kód, který spustil podmínku. Pêi ERRORových pêeruîeních je tato hodnota vrácenou hodnotou kommanda a mûùe být normálnë povaùována za stupeñ chyby. Hodnota SYNTAX interruptu je vùdy arexxový kód.

Tyto proměnné ovîem vznikají i pêi pouhém volání procedûry nebo funkce.

Chci vëdët na co je to dobré
Poznámka: Mohly by se vám hodit funkce ERRORTEXT() a SOURCELINE().

Interrupty jsou velmi potêebné pêi opravë chyb. K tomu patêí informace externích programû o výskytu chyby nebo hláîení dalîích diagnostických dat k ohraniçení problémû. Ukázkový program 15 ukazuje, jak je kommando "message" pêi zjiîtëní chyby posláno na externí host.

Program 15. Interrupt.rexx (Program je nespustitelný !)

```
/* Makro program pro 'mûjedit' */
SIGNAL ON SYNTAX    /* Aktivovat interrupt */
... ( zde následují ostatní pêíkazy)
EXIT

SYNTAX:    /* V pêípadë syntaxní chyby... */
ADDRESS 'mûjedit'
'message' 'error' RC errortext(RC)
EXIT 10
```

Pêíklad pouùití mûùete prostudovat v programu, který se stará o to, aby bylo moùné ukázkové programy pêímo spustit. Tento program se nazývá " Programy.rexx "

1.203 sigl-ukazka

Ukázka pouùití promënné SIGL pêi volání funkce

Promënná SIGL vzniká pêi kaùdém volání funkce a poukazuje na çíslo êádku, z nëhoù byla funkce volána. Asi se zeptáte, na co je dobré vëdët z kterého êádku byla procedûra volána. Ukáùu vám to na pêíkladë:

ARexx umouòuje vrátit kterýkoliv êádek právë bëùícího pêíkazu a to pomoci funkce

SourceLine()

,toho lze vyuùít napêíklad tak, ùe do programu umístíte nëjaká data, napê. slovník:

```
name · jméno
heavy · tëùký
street · ulice
```

Docílíte tím toho, ùe budete mít program i slovník v jednom soubor. Pomocí fnkce

SourceLine()

sice mûùete k tëmto datûm pêistouit, ale nezjistíte na kterém êádku tato data zaçínají. Poçítat s tím, ùe zaçínají stále od stenéhо êádku není pêiliî vhodné, proto zde lze pouùít promënné SIGL, která urçí çíslo êádku. Musí se to vîak provést trochu krkolomnë:

```
POZICE:; ARG P; IF P='' Then Call POZICE(1); Return SIGL+1
```

Tato funkce po zavolání volá jeîtë jednou sama sebe a vrátí çíslo následujícího êádku. V programu pak uù jen staçí napsat:
radek = Pozice() a znáte çíslo êádku na nëmù zaçínají data...
nebo spíîe çíslo êádku, který následuje za funkcí "POZICE:".

1.204 7. kapitola - Syntaxní analýza, Význam a popis

Syntaxní analýza (parsing)

Tuto kapitolu jsem byl nucen praktický celou pêepsat, protoùe byla tak nesrozumitelná, ùe se z ní nedalo nic pouùít. A proto doufám, ùe se mi podaêilo zvolit správný postup a ùe bude pro vás srozumitelná.

Význam syntacní analýzy

Význam syntaxní analýzy je zcela zêejmý, lze s její pomocí velice jednosuîe rozloùit êetëzec na nëkolik çástí a ty pêidëlit jednotlivým promënným. Stejného výsledku by se dalo dosáhnout i s pouùítím êetëzcových funkcí jako

SUBSTR()

,

INDEX()

,
WORD()
,
SUBWORD()
,
POS()
, atd.
Syntaxní analýza je vîak nesrovnatelnë jednoduîîí a efektivnëjîí, obzvláîtë kdyù je êetëzec rozkládán na mnoho menîích. Kdo tomu nechce vëêit, mûùe se pêesvëdçit na jednoduchém
pêíkladë
.

Pouùití syntaxní analýzy

Syntaxní analýza je volána kteroukoliv variantou pêíkazu
PARSE
.
Jednotlivé varianty se liîí pouze vstupním êetëzem, kterému se také êíká "analýzový êetëzec". U nëketrých voleb je dodán automatický, zatímco u posledních dvou voleb VAR a VALUE mûùete rozkládat zcela libovolný êetëzec.
Syntaxní analýzu lze dále volat samostatnými pêikazy pêíkazy
ARG
,

PULL
, které vîak zdrojový êetëzec pêed analýzou pêevedou na velká ↩
písmena.

Nedílnou souçástí syntaxní analýzy je
Îablona
, která je vlastnë tou
nejpodstatnëjîí vëcí, která êídí celý rozklad a pêidëlování çástí promënným.

1.205 Pêíklad...

Pêíklad syntaxní analýzy:

Nëkdy je potêeba dostat z ùetezce "Version: $VER: V1.0 -TP, 24-12-1998", text mezi "$VER:" a çárkou. Pokud by jste chtëli pouùít klasické pêíkazy, museli by jste program napsat takto:

```
/* Parsing 1 */
Text = 'Version: $VER: V1.0 -TP, 24-12-1998'
pocatek = Pos('$VER:',Text)+5 ; konec = Pos(',',Text)
Verze = Substr(Text,pocatek,konec-pocatek)
Say Verze
```

Pouùití pêíkazu

PARSE
se program velice zhednoduší:

```
/* Parsing 2 */
Text = 'Version: $VER: V1.0 -TP, 24-12-1998'
Parse VAR Text '$VER:'verze','
Say Verze
```

A teð si pêedstavte, ùe by jste chtëli rozloùit nëjaký velký êetezec na spoustu malých? Asi by vás to brzy omrzelo...

Jak jistë sami vydíte skrývá se pod syntaxní analýzou velice mocný nástoroj, který vám umoùní velice pohodlnë pracovat s textovými êetëzci.
Mûùete si napêíklad vyzkouîet napsat program, který pêeloùí pêíkaz pro odkaz z formátu AmigaGuide do HTML jazyka.

```
AmigaGuide:   @{" Nazev odkazu " LINK <cíl>}
HTML jazyk:   <A HREF="cíl">Nazev Odkazu</A>
```

Zdánlivë sloùity úkol se s pouùitím syntaxní analýzy stává jednoduchým, proto neváhejte a dobêe prostudujte
pouùití syntaxní analýzy
.

1.206 Popis îablon êídících syntaxní analýzu

Podstatu a základy najdete v nadêazené kapitole
Syntaxní analýza
Î A B L O N Y

Îablona je êetëzec, který se mûùe skládat ze symbolû, êetëzcû,operátorû a závorek. Jejím úkolem je êídit syntaxní analýzu.

Pro pochopení je têeba znát
princip zpracování îablony
a vëdët, ùe
obecnë je kaùdá îablona tvoêena ze dvou objektû, se
Znaçek
a
Cílu
.

Existuje nëkolik druhû îablon, které mûùete samozêejmë vzájemnë kombinovat. Nyní si je blíùe popíîeme:

*
Prosté pêidëlování slov promënným
(Tokenizace)
- zástupný znak "
.
"

*
Vyhledávání vzorû ve zdrojových êetëzcích
(vzorové znaçky)

*
Rozdëlování êetëzcû na definovaných pozicích
(poziçní znaçky)

-
Víceré îablony

1.207 Princip zpracování Îablony bëhem syntaxní analýzy

Zpracování îablony

Nyní jiù víte, co je podstatou
syntaxní analýzy
a nyní si pêiblíùeme
jeji princip, coù vám usnadní pochopení dalîích krokû výuky.

Pro názornost zavedu pojem "kurzor". Tento smyîlený kurzor bude urçovat pozici ve zdrojovém (analyzovaném) êetëzci. Na poçátku syntaxní analýzy je tento kurzor na prvním znaku zdrojového êetëzce. Îablona êídí jeho posun a urçuje která jeho çást bude zkopírovaná a do které proměnné. Je to jako pêikládání çtecí hlavy bëhem kopírovaní z magnetofonového pásku.

Êídící symboly,jenù êídí posun kurzoru se nazývají "
Znaçky
" a proměnné,
kterým jsou pêidëlovaný části zdrojového êetëzce jsou "
Cíle
". Arexxový
interpret çte îablonu postupnë zleva do prava, a podle toho jestli narazí na Znaçku nebo cíl provede pêísluînou opereraci, tj. buð pouhý posun kurzoru nebo zkopírování urçité çásti. A stëjnë jako u magnetofonového pásku lze realizovat posun v obou smërech, ale çíst lze jen zleva do prava a navíc, pokud se kurzor dostane na konec zdrojového êetëzce je zbývajícím proměnným pêidëlen prázdný êetëzec.

Snad je vám tento princip jasný a pokud ne, nezoufejte a pokraçujte ve studiu dál, pozdëji si dáte vîe do souvislosti, zvláîï dûleùité pro vás budou pêíklady na kterých se pêesvëdçíte jak to vîechno doopravdy funguje.

1.208 Znaçky

Význam a popis "Znaçek"

Znaçky êídí pozici "kurzoru", urçují tedy nebo ohraniçují çást zdrojového êetëzce, která bude pêidëlena (zkopírována) proměnné v
îablonë
.

Znaçek existuje nëkolik druhû, se vîemi se podrobnë seznámite v dalîích çástech této kapitoly, zde uvádím pouze struçný souhrn s odkazy na çásti zabývající se tëmito znaçkami:

Vzorové znaçky - êetëzcové tokeny, které jsou vyhledávány ve zdrojovém êetëzci nebo promënné v závorkách jejichù hodnota je vyhledávána. Více viz.
Syntaxní nanlýza podle vzorû
Poziçní znaçky - pêímo êídí pozici "
kurzoru
" ve zdrojovém êetëzci
- Více viz.
Syntaxní analýza êízena poziçními znaçkami
* Absolutní - mëní pozoci "kurzoru" vzhledem k zaçátku ↩ êetëzce
- jsou tvoêeny çíslem nebo promënnou, jenù uvádí pozici. Pêed promënnou musí být navíc znak "=",aby bylo zêejmé, ùe se nejdedná o
Cíl
. Zatímco u çíselného symbolu není tento znak nutný.

* Relativní - mëní pozoci "kurzoru" vzhledem k jeho aktuální pozici
- jsou tvoêeny çíslem nebo promënnou, jenù urçuje posun "kurzoru". Pêed nimi se vùdy musí nacházet znaménka "+" nebo "-", která urçují smër posunu

V
îablonë
lze vîechny druhy znaçek vzájemnë kombinovat, lze tak napêíklad dosáhnout toho, aby dvë cílové promënné (urçené îablonou) obsahovaly totoùnou çást zdrojového êetëzce.

1.209 Cíle

Význam a popis "Cílû"

Cíle jsou názvy promënných (symbolû) jimù bude bëhem
syntaxní analýzy
pêídëlena nëjaká hodnota (çást zdrojového êetëzce) nebo ↩ prázdný êetëzce
pokud na në jiù zdrojový êetëzec "nevyîel" (
kurzor
se dostal na konec
zdrojového êetëzce).

Cíle se narozdíl od
znaçek
nepodílí na zmënë polohy
kurzoru
. Nepêímou
vyjímkou je

tokenizace
, pêi níù je zdrojový êetëzec rozkládán na tokeny,
çili slova. Tokenizace je provádëna jen tehdy, kdyù po cíli hned následuje dalîí cíl. Mezi dvëma cíli tedy není znaçka, která by êídila, která çást zdrojového êetëzce pêíjde pêidëlit prvnímu cíli a která druhému. Arexxový interpret tudíù pêidëlování zdrojového êetëzce ukonçí na konci slova (dáno mezerou; ASCII hodnota 13). A dalîí çást pêidëluje následujícímu cíli...

Cílem je také tzv. "místodrùitel", který je pêedstavován teçkou. Chová se stejné jako kaùdá jiná cílová promënná, které vîak ve skuteçnosti není pêidëlena ùádná hodnota, çímù se uîetêí pár bytû volné paměti.

1.210 Syntaxní analýza skrz tokenizaci

Prosté pêidëlování slov promënným (Tokenizace)

Nejjednoduîîím zpûsobem pouùití syntaxní analýzy je rozklad êetëzce na slova,v takovém pêípadë staçí zadat:

```
PARSE
VAR zdroj PromA PromB PromC
```

Promënná zdroj tedy bude rozdëlena na 3 çásti, promënná 'PromA' bude obsahovat první slovo, 'PromB' druhé slovo a 'PromC' celý netokenizovaný zbytek êetëzce. To znaméná pêedevîím to, ùe bude obsahovat na zaçátku mezeru, protoùe ta následuje po druhém slovu. Vîechny pêedcházející promënné jsou naopak tokenizovány, takùe neobsahují na zaçátku ani na konci ùádné mezery.

Pokud nechcete, aby bylo nëkteré slovo ze zdrojového êetëzce pêidëleno nëjaké promënné, mûùete místo ní pouùít tzv. "mistodrùitele", ten je zastoupen teçkou ("."). Místodrùitel se tedy chová jako promënná, které vîak ve skuteçnosti není pêidëlena ùádná hodnota. Pêedcházející pêíklad by tak vypadal takto:

```
PARSE
VAR zdroj PromA . PromC
```

Mistodrùitele mûùete jeîtë pêidat na konec pêedeîlého pêíkladu promënné 'PromC' tak bude pêidëleno pouze jedno slovo a ne celý zbytek zdrojového êetëzce !

Zhrnutí:
Kdyù bezprostêednë za promënnou v îablonë následuje dalîí promënná, je výchozí êetëzec rozdëlen na slova (pomocí mezer). Na zaçátku a na konci îablony nemají mezery význam. Kaùdé slovo je pêiêazeno jedné promënné. Normálnë dostane poslední promënná netokenizovaný zbytek pûvodního êetëzce, protoùe po ní nenásleduje ùádný symbol. Místodrùitel (.) vede k tomu, ùe promënná s teçkou skonçí, jakmimle se v vstupním proudu vyskytne mezera. Místodrùitelé se chovají jako promënné, i kdyù jim nikdy není pêiêazena hodnota.

Nyní se podíváme na nëkolik pêíkladû:

```
zdroj= 'Jablko Nezralé 15.16 Kç Vyprodáno'

PARSE VAR
Zdroj druh typ cena . stav .
```

Je naprosto zêejmé, ùe kaùdé slovo bude pêidëlëno jedné promënné. V îablonë se nachází dva místodrùitelé '.', první z nich má naprosto zêejmý význam, naharazuje totiù promënnou, které mý být pêiêazen êetëzec "Kç". Druhý místodrùitel vede k tom, ùe pêedchozí promënná (stav) obdrùí tokenizovanou hodnotu. Kdyby byl místodrùitel vynechán, byl by celý zbytek analýzového êetëzce pêiêazen promënné "stav" s pêedcházející mezerou.
Hodnota promënné je v tomto pêípadë 'Vyprodáno', pokud vynecháte '.' na konci îablony, bude mít tato promënná tuto hodnotu: ' Vyprodáno'.

Dalîí pêíklad ukazuje, ùe îablona mûùe obsahovat tûtéù promënnou jako zdroj:

```
 zdroj = "Jen Amiga to dëlá moùné"

                 DO
                 forever

                 PARSE VAR
                 zdroj slovo zdroj

                 IF
                 zdroj =='' THEN LEAVE
     /* Ukonçit, jstliùe nejsou k dispozici dalîí slova */

                 SAY
                 zdroj
   END
```

Z êetëzce "zdroj" je vùdy odêiznuto jedno slovo. Proces je opakován tak dlouho, dokud jiù nemohou být extrahována ùádná dalîí slova.

Výstup programu vypadá takto: (na zaçátku kaùdého êetëzce je mezra !)

```
 Amiga to dëlá moùné
 to dëlá moùné
 dëlá moùné
 moùné
```

1.211 Syntaxní nanlýza podle vzorû

Vyhledávání vzorû ve zdrojových êetëzcích

Pro vyhledávání ve zdrojových êëtëzcích slouùí tzv. "vzorová znaçka" Vzorová znaçka je êetëzcový token libovolné délky, který je bëhem syntaxní analýzy vyhledáván ve zdrojovém êëtëzci.

Lze také pouùít promënnou uzavêenou do kultatých závorek, v tom pêípadë bude vyhledávána její hodnota.

Pêi tomto vyhledávání je rozliîováno mezi malými a velkými písmeny. Tato forma îablony je vîak vhodná pêedevîím v pêípadech, kdy jsou pro vás

dûleùitá data ve zdrojovém êetëzci ohraniçena nëjakými pêesnë danými znaky (napê závorkami).

Po vyhledávání bude "
kurzor
" pêesunut na konec hledaného êetëzce, následujícímu cili tedy bude pêidëlována çást za vzorovou
znaçkou
.

Pokud nebude ve zdrojovém êetëzci vzor nalezen pêesune se kurzor aù na konec êetëzce ! Mûùe se vám tedy snadno stát, ùe zbývající promënné v îablonë obdrùí uù jen prázdný êetëzec. Pokud si tedy nejste jisti výskytem hledaného êetëzce ve zdroji, radëji pouùíjte napê. funkce
POS()
s níù mûùete polohu zjistit jiù pêed samotnou analýzou a ↩
podniknout
pêísluîné kroky.

Pêíklady:

```
/* Vyhledávání êetëzce */
zdroj = 'Amiga (68030/50)'

                PARSE VAR
                zdroj typ '('procesor'/'freq')'

                Say
                'Váî poçítaç' typ 'má procesor' procesor 'o frekvenci' freq 'MHz. ↩
                          '
```

Program vypíše:

```
  -> Váî poçítaç Amiga  má procesor 68030 o frekvenci 50 MHz.
```

```
/* Pouùítí promënné jako vyhledávacího vzoru */
zdroj = 'A:Snídanë B:Obëd C:Veçeêe'
znak  = 'B'

                PARSE VAR
                zdroj (znak)':'druh .
Say druh
```

Program vypíše:

```
  -> Obëd
```

```
/* Pozor na malá a velká písmena */
zdroj = 'Program hledá "REXX", ale pokaùdé jako jiné slovo'

Parse VAR zdroj 'REXX' zbytek ; Say Zbytek
Parse VAR zdroj 'rexx' zbytek ; Say Zbytek
```

Program vypíše:

```
  -> ", ale pokaùdé jako jiné slovo
  -> /* v druhém pêípadë navypíše nic */
```

1.212 Syntaxní analýza êízena poziçními znaçkami

Rozdëlování êetëzcû na definovaných pozicích

Poziçní
znaçky
slouùí ke zmënë polohy "
kurzoru
". Lze tedy pomocí
nich extrahovat çásti zdrojového êetëzce bez ohledu na jeho obsah.

ARexx disponuje dvëma druhy poziçních znaçek:

* Absolutní znaçky
Slouùí ke zmënë pozice
kurzoru
vzhledem k zaçátku zdrojového êetëzce.
Nezáleùí tedy na jeho pêedchozí poloze. Absolutní znaçky mohou být tvoêeny buð çíselným symbolem nebo promënnou, jejiù hodnota mûùe být v rozsahu 1 aù poçet znakû zdrojového êetëzce. Mûùe mít sice vëtší hodnotu, ale nemá to význam, protoùe kurzor zûstane stát vùdy na konci zdrojového êetëzce. Pokud se jedná o promënnou, musí jí vîak pêedcházet znak "=", aby bylo zêejmé, ùe se nejedná o
cílovou promënnou
, ale o
znaçku
.

U çíselného symbolu není nutné tento znak uvádët, ale je to moùné.

Co se týçe samotného prûbëhu
syntaxní analýzy
,je têeba vëdët,ùe cílové
promënné následující za touto znaçkou bude pêidëlena çást zdrojového êetëzce, která zaçíná polohou, kterou znaçka urçuje a konçí podle toho, co následuje za cílovou promënnou:

* Pokud je za
cílovou promënnou
dalîí poziçní znaçka, konçí extrahovaný
êetëzec na pozici, kterou udává.
* Jestliùe následuje dalîí cílová promënná, konçí extrahovany êetëzec na nejbliùîí mezeêe ve zdrojovém êetëzci. Stejnë jako pêi
tokenizaci
.
* A kdyù následuje vzorová znaçka konçí pêidëlování pêed nalezenou nalezenou vzorovou znaçkou nebo na konci zdrojového, kdyù není vzor nalezen.

Pêíklad:

```
/* Ukázka pouùití absolutní poziçní znaçky */
Zdroj = 'Tohle je ukátkový text'
konec = 14
```

```
                PARSE VAR
                Zdroj 7 mezi =konec   /* zde je pouùito obou zpûsobû */

                Say
                mezi

 Program vypíše:
   -> je ukát
```

  * Relativní znaçky

Slouùí ke zmënë pozice kurzoru vzhledem k jeho aktuální poloze a to v obou smërech. Rlativní stejnë jako Absolutní znaçky mohou být tvoêeny çíselným symbolem nebo promënnou, ale v obou pêípadech musí pêed nimi být znak "-" nebo "+", který urçuje zda se jedná o posun vlevo nebo vpravo.

Çíslo udává posun kurzoru ve znacích a mûùe být vëtîí neù je zbývající dëlka êetëzce. Kurzor se totiù zastaví na zaçátku (poloha 1) nebo na konci zdrojového êetëzce.

Prûbëh

```
                syntaxní analýzy
               , respektivë providla platící o dëlce êetëzce,
který bude pêidëlen
                cílové promënne
               ,jsou stejná jako u Abolutních znaçek.

Pêíklad:
 /* Ukázka pouùití obou druhû poziçních znaçek */
 zdroj = 'Amiga je nejlepîí poçítaç na svëtë!'

                PARSE VAR
                zdroj 10 prvni +1 CoJe'!' 1 Kdo .

                Say

                UPPER
               (prvni)Coje 'je' Kdo '!!!'

 Program vypíše:
   -> Nejlepîí poçítaç na svëtë je Amiga !!!
```

V pêedcházejícím pêíkladë bylo pouùito také vzorové znaçky, jejichù pouùití je popsáno v çásti popisující

```
                Syntaxní nanlýza podle vzorû
               .
```

1.213 Víceré îablony

```
               7.3.4  Víceré îablony


   K pêíkazu
                PARSE
```

nebo
ARG
a
PULL
mûùete zadat více îablon. Staçí je
od sebe oddëlit çárkami.

Takto se k více îablonám chovají jednoltivé volby nebo samostatné pêíkazy:

*
ARG
- kaùdé îablonë je pêidëlen jeden z argumentových êetëzecû, ↩
které
byly dány funkci pêi jejím volání k dispozici.

*
PULL
- Kaùdé îablonë je pêidëlen jeden vstup z klávesnice. Uùivatel
tedy musí zadat tolik êádkû, kolik má pêíkaù îablon.

*
EXTERNAL
- kaùdé îablonë je pêidëlen jeden êádek s datového proudu
STDERR.

U ostatních voleb je kaùdé îablonë pêidëlen totoùný zdrojový êetëzec.
U volby
VALUE
pêíkazu
PARSE
není výraz vyhodnocován pro kaùdou îablonu
znovu, ale jen jednou.

1.214 Pêíloha A - Chybová hláîení

Chybová hláîení

Kdyù arexxový interpret zjistí chybu, vrátí chybový kód, který urçuje typ chyby. Normálnë je zobrazen chybový kód, çíslo êádku ve zdrojovém textu, na kterém doîlo k chybë a krátká, doplñující hláîka. Poté je zpracování programu pêeruîeno a êízení je pêedáno na volající úroveñ.

Pokud vîak pouùijete pêikazu
SIGNAL
k aktivování nëkterého arexxového
interruptu, napê. SYNTAX, mûùete chybu odchitit a zpracovat pêímo programem. Nelze takto z pochopitelným dûvodû odchytit chyby 1, 3, 4, 5, 6, 7, 8 10 a 11. O tom, jak lze zbývající pêíkazy odchytit a zpracobat se více dozvíte v kapitole zabívající se
Pêeruîenim
.

Kaùdý chybový kód je spojen s urçitým stupnëm závaùnosti chyby, který je nahláîen volajícímu programu jako primární výsledkový kód. Hodnoty tëchto

výsledkových kódû jsou 5 (nepatrná chyba), 10 (bëùná chyba), a 20 (selhání, velmi tëùká chyba). Chybový kód sám tvoêí sekundární výsledek. Zpûsob dalîího hláîení takového kódu závisí na externím (volajícím) programu.

Napêíklad pokud byl program volán ze Shellu, bude k chybovému hláîení vypsanému ARexxem pêidáno jeîtë jedno, které generuje Amiga DOS a mûùe vypadat takto: Command returned 10/12: Error return from function.

Nyní se mûùete podívat na
seznam vîech ARexxových chyb
.

1.215 Çíselný seznam vîech chyb vracených ARexxem

Chybová hláîení - Popis
Zde si mûùete zvolit çíslo chyby o níù chcete vëdët více informací: ↩

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

>48

Abecední seznam chybových hláîení

Poznámka:

Celá tato çást byla z pûvodní verze manuálu do této podoby pêekonvertována pomocí arexxového programu. Stejnë jako vîechny ostatní abecední seznamy a rozdëlení pêíkazû a funkcí do çástí. Arexxem byly také kontrolovány odkazy. Nyní vidíte, ùe arexxem lze dëlat opravdu mnoho....

1.216 Abecední seznam chybových hláîení

Zkrácený anglický popis	Číslo chyby
Arithmetic conversion error	: Chyba ç.47
Boolean value not 0 or 1	: Chyba ç.46
Clause too long	: Chyba ç.7
Command string error	: Chyba ç.11
Error return from function	: Chyba ç.12
Execution halted	: Chyba ç.2
Expression required	: Chyba ç.45
Extraneous characters	: Chyba ç.35
Function did not return value	: Chyba ç.16
Function not found	: Chyba ç.15
Host environment not found	: Chyba ç.13
Incomplete IF or SELECT	: Chyba ç.29
Insufficient memory	: Chyba ç.3
Invalid argument to function	: Chyba ç.18
Invalid character	: Chyba ç.4
Invalid DO syntax	: Chyba ç.28
Invalid expression	: Chyba ç.41
Invalid expression result	: Chyba ç.44
Invalid keyword	: Chyba ç.33

```
Invalid message packet                : Chyba ç.10
Invalid operand                       : Chyba ç.48
Invalid PROCEDURE                     : Chyba ç.19
Invalid statement in SELECT           : Chyba ç.23
Invalid template                      : Chyba ç.37
Invalid TRACE request                 : Chyba ç.38
Invalid variable name                 : Chyba ç.40
Keyword conflict                      : Chyba ç.36
Label not found                       : Chyba ç.30
Missing or multiple THEN              : Chyba ç.24
Missing or unexpected END             : Chyba ç.26
Missing OTHERWISE                     : Chyba ç.25
Nesting limit exceeded                : Chyba ç.43
Program not found                     : Chyba ç.1
Requested library not found           : Chyba ç.14
Required keyword missing              : Chyba ç.34
Symbol expected                       : Chyba ç.31
Symbol mismatch                       : Chyba ç.27
Symbol or string >65535 characters    : Chyba ç.9
Symbol or string expected             : Chyba ç.32
Unbalanced parenthness                : Chyba ç.42
Undiagnosed internal error            : Chyba >48
Unexpected BREAK or LEAVE or ITERATE  : Chyba ç.22
Unexpected ELSE or OTHERWISE          : Chyba ç.21
Unexpected WHEN or THEN               : Chyba ç.20
Uninitialized variable                : Chyba ç.39
Unmatched quote                       : Chyba ç.5
Unrecognized token                    : Chyba ç.8
```

```
                Unterminated comment                    : Chyba ç.6

                Wrong number of arguments               : Chyba ç.17
```

1.217 Popis chyby Program not found

```
                Chyba ç.1: Program not found

  Çeský význam:  Program nenalezen
   Chybový kód:  5  (WARN - nepartná chyba)

   Ke vzniku této chyby mûùe vést ze dvou dûvodû.

   * Pokud zpustíte program RX <jméno programu> (bez celé cesty) je program
hledán  nejprve v aktuálním adresáêi a to s pêíponou .rexx i bez ní a pokud
není program nalezen je totéù jeîtë hledáno v adresáêi REXX:.

   * Aby vîak arexxový interpret uznal nalezený textový soubor jako
arexxový  program  musí  kaùdý program zaçínat komentáêem (/*...*/).  Pokud
ne, dojde také ke vzniku této chyby

  Tato chyba je zjiîtëna externím rozhraním (jiù pêed spuîtëním programu) a
nemûùe být tudíù zachycena ùádným
                interruptem
               .
```

1.218 Popis chyby Execution halted

```
                Chyba ç.2: Execution halted

  Çeský význam:  Provádëní programu zastaveno
   Chybový kód:  10  (ERROR - bëùná chyba)

    Bëh  programu  byl  pêeruîen  klávesami  CTRL-C nebo externím commandem
(pêíkazem) HI.

   Tato  mûùe být odchycena,  kdyù  pêíkazem
                SIGNAL
                aktivován HALT nebo
BREAK_C interrupt.
```

1.219 Popis chyby Insufficient memory

```
                Chyba ç.3: Insufficient memory

  Çeský význam:  Nedostatek pamëti
   Chybový kód:  20  (FAILURE - selhání, velmi váùná chyba)

    Interpreter  nemohl  pro  naçtení  a  spuîtëní  zajistit dostatek volné
```

pamëti.

Pamët je vyhrazena jiù pêi spuîtëní programu, tato chyba tudíù nemûùe být odchycena ùádným
interruptem
.

1.220 Popis chyby Invalid character

Chyba ç.4: Invalid character

Çeský význam: Neplatný znak
Chybový kód: 10 (ERROR - bëùná chyba)

V programu byl objeven êídící znak. Êídící kódy mohou být v programu pouùity, napêíklad v êetëzci, ale musí být definovány v hexadecimálním nebo binárním êetëzci.

Tato chyba je zjiîtëna externím rozhraním (jiù pêed spuîtëním programu) a nemûùe být tudíù zachycena ùádným
interruptem
.

1.221 Popis chyby Unmatched quote

Chyba ç.5: Unmatched quote

Çeský význam: Divný poçet uvozovek (Neuzavêené uvozovky)
Chybový kód: 10 (ERROR - bëùná chyba)

Interpret zjistil v souboru lichý poçet uvozovek (nebo apostrofû) ohraniçující êetëzce.

Ke vznikû chyby nemusí nëkdy dojít, protoêe ARexx dovoluje napsat êetëzce pêes více êádkû, êetëzce také mohou být zadány bez uvozovek, ale pêedevîím se mûùe stát, ùe budou za chybëjící ukonçovací zank povaùovány uvozovky v êetëzci. Program tak bude po syntaxní stránce zcela správný, to vîak neznamená, ùe bude správný i po funkçní stránce. Mûùete tak zbyteçnë dlouho pátrat po takto jednoduché pêíçinë velkých problémû...

Tato chyba je zjiîtëna externím rozhraním (jiù pêed spuîtëním programu) a nemûùe být tudíù zachycena ùádným
interruptem
.

1.222 Popis chyby Unterminated comment

Chyba ç.6: Unterminated comment

Çeský význam: Neukonçený komentáê

Chybový kód: 10 (ERROR - bëùná chyba)

Ke kaùé dvojicí znakû "/*" musí v programu existovat také odpovídající "*/". Pêiçemù komentáêe mûùou být klidnë vnoêeny do sebe.

Tato chyba je zjiîtëna externím rozhraním (jiù pêed spuîtëním programu) a nemûùe být tudíù zachycena ùádným
interruptem
.

1.223 Popis chyby Clause too long

Chyba ç.7: Clause too long

Çeský význam: Pêíliî dlouhý êádek
Chybový kód: 10 (ERROR - bëùná chyba)

Êadek byla pro interní buffer (zásobník) moc dlouhý, musíte jej tedy rozdëlit na nëkolik kratîích.

Tato chyba je zjiîtëna externím rozhraním (jiù pêed spuîtëním programu) a nemûùe být tudíù zachycena ùádným
interruptem
.

1.224 Popis chyby Unrecognized token

Chyba ç.8: Unrecognized token

Çeský význam: Neplatný token (znak)
Chybový kód: 10 (ERROR - bëùná chyba)

V programu byl nalezen znak jenù nelze pouùít v názvu promënné, jako operátor ani nemá ùádný speciální význam. Tyto znaky mohou být pouùity jen v êetëzci, musí být tedy uzavêeny v apostrofech nebo uvozovkách.
Vîechny znaky i jejich význam je popsán v kapitole
tokeny
.

Tato chyba je zjiîtëna externím rozhraním (jiù pêed spuîtëním programu) a nemûùe být tudíù zachycena ùádným
interruptem
.

1.225 Popis chyby Symbol or string >65535 characters

Chyba ç.9: Symbol or string >65535 characters

Çeský význam: Symbol (promënná) nebo êetëzec je moc dlouhý
Chybový kód: 10 (ERROR - bëùná chyba)

Velikost êetëzce nebo promënné, mûùe být maximálnë 65535 znakû.

Tato chyba mûùe být odchycena SYNTAXním
interruptem
,který lze aktivovat
pêíkazen
SIGNAL
.

1.226 Popis chyby Invalid message packet

Chyba ç.10: Invalid message packet

Çeský význam: Neplatný zprávový paket
Chybový kód: 10 (ERROR - bëùná chyba)

Ve zprávovém balíku, který byl poslán na residentní arexxový proces, byl objeven neplatný akçní kód. Paket byl nezpracovaný pêedán zpët.

Tato chyba je zjiîtëna externím rozhraním (jiù pêed spuîtëním programu) a nemûùe být tudíù zachycena ùádným
interruptem
.

1.227 Popis chyby Command string error

Chyba ç.11: Command string error

Çeský význam: Chyba v kommandovém (pêíkazovém) êetëzci
Chybový kód: 10 (ERROR - bëùná chyba)

Kommandový êetëzec nemohl být zpracován.

Tuto chybu negeneruje pêímo interpret ARexxu, ale jiný program s arexxovým portem, na který byl poslán pêíkaz se kterým si program nedokázal poradit. Pomoc tedy musíte hledat v dokumentaci k programu...

Tato chyba je zjiîtëna externím rozhraním (jiným programem) a nemûùe být tudíù zachycena ùádným
interruptem
.

1.228 Popis chyby Error return from function

Chyba ç.12: Error return from function

Çeský význam: Funkce vrátila chybu
Chybový kód: 10 (ERROR - bëùná chyba)

Externí funkce nebo jiný program s arexxovým portem vrátil chybový kód rûzný od nuly, aniù by blíùe specifikoval její pêíçinu. Pêesvëdçte se pêedevîím, zda byly zadány správnë argumenty.

1.229 Popis chyby Host environment not found

Chyba ç.13: Host environment not found

Çeský význam: Hosttiteslké prostêedí (message-port) nenalezeno
Chybový kód: 10 (ERROR - bëùná chyba)

Message-port odpovídající zadanému adresovému êetëzci nebyl nalezen. Ujistëte se, zda je daný externí host aktivní, lze tak uçinit napê. funkcí

SHOW()
.

1.230 Popis chyby Requested library not found

Chyba ç.14: Requested library not found

Çeský význam: Poùadovaná knihovna nenalezena
Chybový kód: 10 (ERROR - bëùná chyba)

Funkcí

ADDLIB()

lze do tzv. ressource seznamu pêidat knihovnu. Tyto knihovny v tomto seznamu jsou postupnë (podle svých priorit) prohledávány, pokud interpret narazí na funkci (pêíkaz), která je mu neznámá. Aù v této fázi se testuje zda zadaná knihovna vûbec existuje. Chyba tedy mûùe vzniknout, aù pozdëji bëhem vykonávání programu.

Zrádná je ta skuteçnost, ùe chybu mûùe zpûsobit jiná knihovna, neù ta která obsahuje pêíkaz na kterém doîlo k chybë, protoùe prostë byla prohledávána dêíve. Taková chyba pak mûùe být tëùko objevitelná a mûùe k ní dojít têeba aù na jiném poçítaçí, kde daná knihovna není.
Jeîtë horîí vîak je to, ùe

seznam knihoven

je zpravován exetrním

prostêedím (interpretem), pêiêazením neexistující knihovny v jednom programu tak pêestanou fungovat vîechny ostatní !!!
Davejte si na to pozor a pamatujte, ùe nejlepîí obranou je prevence!

Pokud tedy k této chybë dojde zkontrolujte zda vîechy potêebné knihovny existují a zda jsou funkcí ADDLIB() správnë pêidány do seznamu knihoven.

1.231 Popis chyby Function not found

Chyba ç.15: Function not found

Çeský význam: Funkce nenalezena
Chybový kód: 10 (ERROR - bëùná chyba)

K této chybë dojde pokud arexxový interpret narazý na funkci, kterou nenajde v ùádné z knihoven v
seznamu knihoven
ani ji nezná aktuální
message-port, çili externí program.

Ujistëte se, zda jsou odpovídající knihovny funkcí v seznamu knihoven, zda jste zadaly správnë název portu u pêíkazu ADDRESS a nebo zda je vûbec název funkce správný...

1.232 Popis chyby Function did not return value

Chyba ç.16: Function did not return value

Çeský význam: Funkce nevrátila hodnotu
Chybový kód: 10 (ERROR - bëùná chyba)

Byla volána funkce, která nevrátila výsledkový êetëzec, ani nenahlásila chybu, pêestoùe to od ní bylo oçekáváno. Obvyklou pêíçinou je pêíkaz

RETURN
(nebo
EXIT
) zadaný bez parametrû. Nejlepîí je vùdy psát jako argument k pêíkazu RETURN alespoñ 0.

Ujistëte se, zda byla funkce (procedûra) správnë naprogramována nebo ji zavolejte pêíkazem
CALL
, který nevyùaduje návratovou hodnotu.

1.233 Popis chyby Wrong number of arguments

Chyba ç.17: Wrong number of arguments

Çeský význam: Îpatný poçet argumentû
Chybový kód: 10 (ERROR - bëùná chyba)

Byla volána funkce, která oçekává jiný poçet argumentû. Vëtîina funkcí poùaduje podle situace rûzný poçet argumentû, nëkteré volitlené lze nëkdy vypustit.

Zkontrolujte zda je funkci pêedáván správný poçet argumentû.
Pro kontrolu mûùete pouùít
Výpis vîech funkcí
.

1.234 Popis chyby Invalid argument to function

```
                Chyba ç.18: Invalid argument to function

  Çeský význam:  Neplatný argument pro funkci
   Chybový kód:  10  (ERROR - bëùná chyba)

    Funkci byl pêedán nevhodný argument, nejçastëji se jedná o zámënu çísla
êetëzcem nebo pêedání neinicializované proměnné v dûsledku pêeklepu, atd.

   Zkontrolujte zda jsou funkci pêedávány správné arhumenty.
   Pro kontrolu mûùete pouùít
                Výpis vîech funkcí
               .
```

1.235 Popis chyby Invalid PROCEDURE

```
                Chyba ç.19: Invalid PROCEDURE

  Çeský význam:  Neplatný PROCEDURE
   Chybový kód:  10  (ERROR - bëùná chyba)

    Pêíkaz
                PROCEDURE
                lze pouùít  pouze v podprogrmu (
                interní funkce
               ).
Nejçastëjîí pêíçinou  vzniku  této  chyby je  chybëjící  pêíkaz
                EXIT
                nebo

                RETURN
               ,  protoùe  tak  se  mûùe  stát,  ùe se nachází dva pêíkazy  ↩
                  PROCEDURE
zasebou, coù je nepêípustné.
```

1.236 Popis chyby Unexpected WHEN or THEN

```
                Chyba ç.20: Unexpected WHEN or THEN

  Çeský význam:  Neoçekávané THEN nebo WHEN
   Chybový kód:  10  (ERROR - bëùná chyba)

   Pêíkaz
                WHEN
                nebo sub-pêíkaz THEN byl vyvolán mimo platný kontext.

 *  Pêíkaz WHEN je platný jen  uvnitê oblasti
                SELECT
               .
 *  THEN musí následovat jako dalîí sub-pêíkaz po
                IF
```

```
                nebo
                WHEN
               .
```

1.237 Popis chyby Unexpected ELSE or OTHERWISE

```
                Chyba ç.21: Unexpected ELSE or OTHERWISE

  Çeský význam:  Neoçekávané ELSE nebo OTHERWISE
   Chybový kód:  10  (ERROR - bëùná chyba)

    Pêíkaz
                ELSE
                nebo
                OTHERWISE
                byl nalezen mimo platný kontext.

 * Pêíkaz OTHERWISE je platný jedinë uvnitê oblasti
                SELECT
               .
 * ELSE je platný jen jako následovník THEN-vëtve uvnitê
                IF
               -ové oblasti.
```

1.238 Popis chyby Unexpected BREAK or LEAVE or ITERATE

```
                Chyba ç.22: Unexpected BREAK or LEAVE or ITERATE

  Çeský význam:  Neoçekávané BREAK, LEAVE nebo ITERATE
   Chybový kód:  10  (ERROR - bëùná chyba)

   Pêíkaz
                BREAK
                je platný jen uvnitê
                DO
                oblasti nebo v
                INTERPRET
               ovaném
êetëzci.   Pêíkazy
                LEAVE
                 a
                ITERATE
                 jsou  platné  jen  v  iterativní
(opakujcící se) DO oblasti.
```

1.239 Popis chyby Invalid statement in SELECT

```
                Chyba ç.23: Invalid statement in SELECT

  Çeský význam:  Neplatný pokyn v bloku SELECT
```

```
   Chybový kód:  10  (ERROR - bëùná chyba)

   Uvnitê  oblasti
                SELECT
                 byl  objeven  neplatný pokyn.  Platné jsou Jen
pokyny
                WHEN
                , THEN,
                OTHERWISE
                , pokud  se nedíváme na podmínëné pokyny po
klausulích THEN nebo OTHERWISE.

   Chybu mûùe zpûsobit napêíkûad také chybná (neukonçená)
                DO
                oblast.
```

1.240 Popis chyby Missing or multiple THEN

```
                Chyba ç.24: Missing or multiple THEN

  Çeský význam:  Chybëjící nebo nadbiteçné THEN
   Chybový kód:  10  (ERROR - bëùná chyba)

   Programem  oçekávaná  klausule  THEN nebyla nalezena, nebo bylo zjiîtëno
dalîí THEN, poté co jiù jedno bylo provedeno.

   THEN je souçástí pêíkazû
                IF
                nebo
                WHEN
                .
```

1.241 Popis chyby Missing OTHERWISE

```
                Chyba ç.25: Missing OTHERWISE

  Çeský význam:  Chybëjící OTHERWISE
   Chybový kód:  10  (ERROR - bëùná chyba)

   Ùádná  z  klausulí
                WHEN
                v  oblasti
                SELECT
                nebyla  úspëîná  a  chybí
klausule
                OTHERWISE
                .

   Klausule  OTHERWISE  není  nutná, pokud je splnëna nëkterá z pêedchozích
podmínek,  coù  vîak  nelze  vùdy  zajistit, proto je vhodné OTHERWISE vùdy
pouùívat alespoñ s argumentem NOP (ùádný pêíkaz).
```

1.242 Popis chyby Missing or unexpected END

```
                Chyba ç.26: Missing or unexpected END

  Çeský význam:  Chybìjící  nebo neoçekávaný END
   Chybový kód:  10  (ERROR - bëùná chyba)

   Pêíkaz
                END
                nelze  nikdy  pouùít  samostatnë  musí  se vùdy nacházet aù
na konci
                DO
                nebo
                SELECT
                oblasti. Ke kaùdému pêíkazu DO nebo SELECT tedy
musí pêísluîet jedno END.

  Pêíçinou  této  chyby  mûùe  být  také chybìjící apostrof (nebo uvozovka)
na konci êetëzce, pêíkaz END tak mûùe být povaùován za souçást êetëzce.
```

1.243 Popis chyby Symbol mismatch

```
                Chyba ç.27: Symbol mismatch

  Çeský význam:  Chybný symbol
   Chybový kód:  10  (ERROR - bëùná chyba)

   Argument  zadaný  v pêíkazu
                END
               ,
                ITERATE
               ,
                LEAVE
                 neodpovídá  názvu
indexní promënné uvedené v DO oblasti.

   Ujistëte  se, zda jsou aktivní smyçky správnë vnoêeny a zda jsou správnë
pojmenovány jejich indexní promënné.
```

1.244 Popis chyby Invalid DO syntax

```
                Chyba ç.28: Invalid DO syntax

  Çeský význam:  Chybná syntaxe pêíkazu DO
   Chybový kód:  10  (ERROR - bëùná chyba)

   Podívejte se  na  popis  pêíkazu
                DO
                 a zkontrolujte  zda máte spravnë
zadány   vîechny  jeho  moùné  argumenty.  Pokud  pouùíváte  jako  nëkterý
z parametrû  pro pêíkaz  DO  promënnou nebo êetëzec, zkontrolujte správnost
jejich hodnoty.
```

1.245 Popis chyby Incomplete IF or SELECT

```
                Chyba ç.29: Incomplete IF or SELECT

  Çeský význam:  Nekompletní pêíkaz IF nebo SELECT
   Chybový kód:  10  (ERROR - bëùná chyba)

    V oblastech
                SELECT
                nebo
                IF
                musí vùdy za klíçovým slovem THEN, popê.
ELSE  nebo OTHERWISE následovat libovolný pokyn nebo více pokynû uzavêených
v
                DO
                 oblasti.  Pokud je pouùití pêíkazu neùádoucí, staçí pouùít  ↩
                    argument
NOP (ùádný pêíkaz).
```

1.246 Popis chyby Label not found

```
                Chyba ç.30: Label not found

  Çeský význam:  Skoková znaçka nenalezena
   Chybový kód:  10  (ERROR - bëùná chyba)

   Skoková znaçka, která byla zadána v pêíkazu
                SIGNAL
                nebo na kterou bylo
odkázáno  implicitnë  aktivním  interruptem,  nebyla  ve  zdrojovém  textu
programu nalezena.

   Na  Skokové  znaçky, které byly  aktivovány  pêíkazem
                INTERPRET
                 nebo
dynamicky  pêi  dialogovém  vstupu,  nelze pouùít !!!  Lze se odkazovat jen
na znaçky definované ve zdrojovém textu programu.
```

1.247 Popis chyby Symbol expected

```
                Chyba ç.31: Symbol expected

  Çeský význam:  Byl oçekáván symbol
   Chybový kód:  10  (ERROR - bëùná chyba)

   Token, který není
                symbolem
               , byl objeven na místë, kde má být symbolový
token. Po pêíkazech
                DROP
               ,
                END
```

```
                ,
                 LEAVE
                ,
                 ITERATE
                 smí následovat
jen symbolový token.  Ke vzniku této chby dojde i kdyù chybí.
```

1.248 Popis chyby Symbol or string expected

```
                 Chyba ç.32: Symbol or string expected

  Çeský význam:  Byl oçekáván symbol nebo êetëzec
   Chybový kód:  10  (ERROR - bëùná chyba)

   Neplatný token, byl oçekáván jedinë
                 symbol
                 nebo
                 êetëzec
                .
```

1.249 Popis chyby Invalid keyword

```
                 Chyba ç.33: Invalid keyword

  Çeský význam:  Neplatné klíçové slovo
   Chybový kód:  10  (ERROR - bëùná chyba)

   Klíçové slovou, pouùito v této sitaci není správné.

   Popis klíçových slov je u abecedního seznamu
                 Pêíkazû
                 a
                 Funkcí
                .
```

1.250 Popis chyby Required keyword missing

```
                 Chyba ç.34: Required keyword missing

  Çeský význam:  Chybí klíçové slovo
   Chybový kód:  10  (ERROR - bëùná chyba)

   Souçásti  nëkteých  pêíkazû  nebo  funkcí  je  kliçové slovo upêesñující
význam argumentu.  V tomto pêípadë chybí nebo je nevhodnë pouùito.

   K této chybë napê. dojde, kdyù za pêíkazem
                 SIGNAL
```

ON nenásleduje
interruptové klíçové slovo (napê. SYNTAX).

1.251 Popis chyby Extraneous characters

Chyba ç.35: Extraneous characters

Çeský význam: Nadbyteçné znaky na konci pêíkazu
Chybový kód: 10 (ERROR - bëùná chyba)

V programu jsou zadány znaky, které na tomto místë nelze pouùít (çárka, dvojteçka), jsou zde nadbiteçné...

Mûùe se jednat napêíklad o nevhodnë pouùité speciální znaky, pêedevîím oddëlovaçe.

1.252 Popis chyby Keyword conflict

Chyba ç.36: Keyword conflict

Çeský význam: Konflikt klíçových slov
Chybový kód: 10 (ERROR - bëùná chyba)

Pêíkazová klausule obsahuje dvë klíçová slova, která se navzájem vyluçují, nebo je klíçové slovo zadáno dvakrát ve stejném pêíkazu.

1.253 Popis chyby Invalid template

Chyba ç.37: Invalid template

Çeský význam: Neplatná îablona
Chybový kód: 10 (ERROR - bëùná chyba)

Îablona, zadaná k pêíkazu
PARSE
,
ARG
nebo
PULL
není správnë
sestavena.

Pokud nerozumíte tvorbë
îablon
a jejich podstatë, mûùete prostudovat
kapitolu
Syntaxní analýza
.

1.254 Popis chyby Invalid TRACE request

```
                Chyba ç.38: Invalid TRACE request

  Çeský význam:  Neplatný poùadavek TRACE
   Chybový kód:  10  (ERROR - bëùná chyba)

   Parametr k pêíkazu
                TRACE
                 nebo integrované funkci funkci
                TRACE()
                je
chybný.
```

1.255 Popis chyby Uninitialized variable

```
                Chyba ç.39: Uninitialized variable

  Çeský význam:  Neinicializovaná proměnná
   Chybový kód:  10  (ERROR - bëùná chyba)

   K této chybë dojde pêi pokusu o pouùití neinicializované proměnné, pokud
je aktivován
                interrupt
                 NOVALUE (
                SIGNAL
                ON NOVALUE), který si pouùití
neinicializovaných proměnných hlídá.

 Poznámka:
                Symboly
                (proměnné) jsou inicializovány jakmile jim je poprvé
             pêidëlena hodnota.
```

1.256 Popis chyby Invalid variable name

```
                Chyba ç.40: Invalid variable name

  Çeský význam:  Neplatné jméno proměnné
   Chybový kód:  10  (ERROR - bëùná chyba)

   Pokus o pêiêazení hodnoty pevnému
                symbolu
               .

   Tato  chyba  se  objevuje  jen velice vzácnë, protoùe interpret vëtîínou
(chybnë) povaùuje  pêiêazovací
                výraz
                za
                logický
               , pêipadnë vyçíslí jinou
chybu, napêíklad
```

47
.

1.257 Popis chyby Invalid expression

Chyba ç.41: Invalid expression

Çeský význam: Neplatný výraz
Chybový kód: 10 (ERROR - bëùná chyba)

Pêi vyhodnocení
výrazu
byla zjiîtëna chyba.

Ujistëte se, ùe ke kaùdému
operátoru
je zadán správný poçet operandû a
ùe ve výrazu nejsou znaky navíc. Dále zkontrolujte jestli obsah proměnných
odpovídá operacím, které jsou s nimi prováděny (násobení êetëzcu, atd.).
Chybu mûùe zpûsobit také chybějící apostrof u
êetëzce
, jeho znaky tak
mohou být povaùovány za matematické operátory.

Tato chyba je zjiîtëna jen ve výrazech, které byly skuteçnë vyhodnoceny.
Pêi výrazech v
klausulích
, které jsou pêeskoçeny, nedojde k vyhodnocení.
Chyba se tak nemusí projevit hned !

1.258 Popis chyby Unbalanced parenthness

Chyba ç.42: Unbalanced parenthness

Çeský význam: Chybný poçet závorek
Chybový kód: 10 (ERROR - bëùná chyba)

V programu se nachází výraz, ve kterém se liîí poçet otevêených závorek
od uzavírajících. Ke kaùdé otevêené závorce musí existovat uzavírající.

Na rozdíl od chyb ç. 5 se tato chyba objeví aù v okamùiku vyçislování
výrazu a lze ji tedy zachytit SYNTAXním
interruptem
.

Viz. také pêíkaz
SIGNAL
.

1.259 Popis chyby Nesting limit exceeded

Chyba ç.43: Nesting limit exceeded

Çeský význam: Hranice vnoêení byla pêekroçena (zaplnën vnitêní zásobník)
Chybový kód: 10 (ERROR - bëùná chyba)

Poçet podvýrazû ve
výrazu
pêesahuje únosné hranice. Vnitêní zásobník
dokáùe zpracovat pouze 32 urovní. Zjednoduîte výraz rozloùením na více
menîích výrazû.

Bliùîí popis naleznete u
Popisu tvorby výrazû pro podmínky
.

1.260 Popis chyby Invalid expression result

Chyba ç.44: Invalid expression result

Çeský význam: Neplatný výsledek výrazu
Chybový kód: 10 (ERROR - bëùná chyba)

Výsledek
výrazu
byl ve svém kontextu neplatný.

K této chybë mûùe dojít napêíklad, kdyù je inkrement nebo hraniçní
hodnota v
DO
pêíkazu zadána jako výraz jehoù výsledkem je
êetëzec
, atd.

1.261 Popis chyby Expression required

Chyba ç.45: Expression required

Çeský význam: Je oçekáván výraz
Chybový kód: 10 (ERROR - bëùná chyba)

V programu byl vynechán
výraz
, který je v daném kontextu nezbytný.

Napêíklad po pêíkazu
SIGNAL
musí následovat výraz, pokud nenásleduje
klíçové slovo ON nebo OFF. U pêikazu
IF
musí zase následovat výraz

vracející
booleovskou hodnotu
, atd.

1.262 Popis chyby Boolean value not 0 or 1

Chyba ç.46: Boolean value not 0 or 1

Çeský význam: Booleovská hodnota není 0 nebo 1
Chybový kód: 10 (ERROR - bëùná chyba)

Nëkteré pêíkazy nebo funkce vyùadují jako vstupní argument
booleovskou
hodontu, çili 1 nebo 0.
Výraz
vîak vrátil hodnotu rûznou od 0 nebo 1.

Více o logických výrazech naleznete u
Popisu tvorby výrazû pro podmínky
.

1.263 Popis chyby Arithmetic conversion error

Chyba ç.47: Arithmetic conversion error

Çeský význam: Chybná aritmetická konverze
Chybový kód: 10 (ERROR - bëùná chyba)

Nenumerický operand (êetëzec) byl pouùit v operaci, která vyùaduje numerické operandy. K této chybë dojde i v pêípadë neplatného hexadecimálního nebo binárního êetëzce.

Nejçastëjîí pêíçinou této chyby je neinicializovaná promënná, která má spravnë obsahovat numerickou hodnotu, pokud nelze takovéto situaci z nëjakého dûvodu zabránit, lze této chybë pêedejít pomoci
interruptu
NOVALUE, který odchytí pouùití neinicializované promënné.

1.264 Popis chyby Invalid operand

Chyba ç.48: Invalid operand

Çeský význam: Neplatný operand
Chybový kód: 10 (ERROR - bëùná chyba)

Operand byl pro zamýîlenou operaci neplatný.

K této chybë dochází nejçastëji pêi pokusu o dëlení nulou nebo pêi pouùití zlomku jako exponentu v mocninë.

Vëtîina ostatních aritmetických chyb je hláîena jako chyba ç.
47
.

1.265 Popis chyby Undiagnosed internal error

Chyba >48: Undiagnosed internal error

Çeský význam: Neurçitelná (nezjistitelná) vnitêní chyba

K této "chybë" dojde jen naprosto vyjímençnë. Më se ji podaêilo dosáhnout jen jednou a to se Arexxem zabývám uù 2 roky. A to právë pêi odlaðování pomocného programu pro tento manuál, chybové hláîení, které vypsala procedûra SYNTAX znëlo takto:

Chyba çíslo 1744830480 -> Undiagnosed internal error

Nevím, jak vám poradit,v pêípadë vzniku této chyby. Snad "Hodnë îtëstí !"

1.266 Pêíloha B - Kommandové pomocné programy

Kommandové pomocné programy

ARexx nabízí êadu pomocných kommandových programû s rûznými êídícími funkcemi. Tyto programy se nacházejí v adresáêi "sys:rexxc/". Pêitom se jedná o spustitelné programy, které mohou být volány ze Shellu a jsou pouùitelné pouze tehdy, kdyù je aktivní residentní arexxový proces, tedy kdyù je spuîtën "RexxMast". Zde je jejejich výpis:

HI
(Halt - Interrupt)

RX
(Rexx eXecute)

RXC
(ReXx Close)

RXLIB
(ReXx LIBrary

RXSET
(ReXx SET)

TCC
(TraCe Close)

```
TCO
 (TraCe Open)

TE
 (Trace End)

TS
 (Trace Start)

WAITFORPORT
```

1.267 Popis pêíkazu HI (Halt - Interrupt)

```
Halt - Interrupt

HI
```

Nastaví globální halt-poznávací znamení, coù znamená, ùe vîechny bëùící arexxové programy obdrùí externí poùadavek halt. Kaùdý program je okamùitë opuîtën, pokud není pro program aktivován HALT-interrupt. Halt poznávací znamení nezûstane nastaveno, nýbrù je okamùitë poté, co vîechny programy obdrùely halt poùadavek, smazáno.

1.268 Popis pêíkazu RX (Rexx eXecute)

```
Rexx eXecute

RX <program> [<argumenty>]
```

Spustí zadaný arexxový program. Pokud není zadaná celá cesta k programu, je zadaný název hledán v aktuálním adresáêi, a poté v adresáêi "Rexx:". Název je hledán z koncovkou ".rexx" i bez ní, ale lepîí je koncovku vùdy pouùívat, uù jenom pro pêehlednost.

Volitelné argumentové êetëzce jsou pêevedeny dále na program.

1.269 Popis pêíkazu RXC (ReXx Close)

```
ReXx Close

RXC
```

Uzavírá residentní proces. Vîeobecnë pêístupný "REXX"-port je bez váhání uzavêen. Residentní proces je opuîtën, jakmile je uzavêen i poslední arexxový program. Po takovéto ùádosti o ukonçení nemohou jiù být startovány dalîí programy.

1.270 k-rxlib

```
                ReXx LIBrary

RXLIB [<jméno> <priorita> [<offset> <verze>]]

   Tento   program   slouùí  k  pêidání nových  knihoven nebo
                host
               -portû
do
                seznamu  knihoven
               .  Syntaxe a význam je shodný s
                integrovanou funkcí

                ADDLIB()
               .

   <jméno> je název pêidávané knihovny (v LIBS:  adresáêi) nebo host-portu.
U jmen je rozliîováno mezi psaním velkých a malých písem

   Priorita  mûùe  nabývat  hodnot  -100  aù  100.  Podle priority jsou pak
pêi
                hledání funkcí
                nëkteré knihovny upêednostñovány.

   Nepovinné parametry <offset> a <verze> platí jen pro knihovny.  <offset>
je  celoçíselná  vzdálenost  od "query" vstupního bodu knihovny (obvykle se
udáva  -30).   A  <verze>  je také celé çíslo, které udává, jakou minimální
verzi knihovny pouùít.

  Pokud  zadáte  pouze  RXLIB  bez  jakýchkoliv  parametrû,  vypíše program
vîechny momentálnë spravované knihovny a host-porty.

Pêíklad:
 rexxreqtools.library (library)
 rexxsupport.library (library)
 rexxarplib.library (library)
 REXX (host)
```

1.271 Popis pêíkazu RXSET (ReXx SET)

```
                ReXx SET

RXSET [<jméno> [[=] <hodnota>]]

   Pêipojí  pár (<jméno>, <hodnota>) do
                clipboardu
               . U jmenných êetëzcû je
rozliîováno  mezi  velkými  a  malými písmeny.  Kdyù jiù pár stejného jména
existuje,  je  jeho  hodnota  nahrazena  novým  êetëzcem (hodnotou).  Je-li
zadáno jméno bez hodnoty, je zápis ze seznamu odstranën.

   Je-li zadán RXSET bez argumentû, jsou vîechny poloùky vypsány.
```

Tento program je shodný s
integrovanou funkcí

SETCLIP()
.

1.272 Popis pêíkazu TCC (TraCe Close)

TraCe Close

TCC

Uzavírá globální monitorovací konzolu, jakmile není od ùádného aktivního programu pouùívána. Çekající poùadavky na çtení musí být zodpovëzeny (stiskem ENTER), neù mûùe být konzola zavêena.

Podrobnosti nalezeznete v kapitole zabývající se
monitorováním
.

1.273 Popis pêíkazu TCO (TraCe Open)

TraCe Open

TCO

Otevêe globální monitorovací konzolu. Výstupní data pro monitorování vîech aktivních programû jsou automaticky pêevedeny na novou konzoli. Uùivatel mûùe pozmënit velikost a pozici okna konzoly, a mûùe okno konzoly zavêít pêíkazem
TCC
.

Podrobnosti nalezeznete v kapitole zabývající se
monitorováním
.

1.274 Popis pêíkazu TE (Trace End)

Trace End

TE

Smaùe znamení pro globální monitorování. Tím je
monitorovací
reùim
pro vîechny aktivní arexxové programy nastaven na OFF.

1.275 Popis pêíkazu TS (Trace Start)

```
                Trace Start

TS

   Startuje
                 interaktivní monitorování
                 nastavením externího monitorovacího
znamení.  Tím  jsou  vîechny arexxové programy pêivedeny do interaktivního
monitorovacího  stavu.  Programy  zaçínají  vytváêet  monitorovací  data a
zastaví  se po pêíîtím pokynu.  Tento pêíkaz je potêeba, kdyù chcete získat
kontrolu  nad  programy,  které  se  nacházejí  v nekoneçných smyçkách nebo
vykazují  jiné  nezvyklé  chybové  symptomy.  Monitorovací znamení zûstává
nastaveno  tak  dlouho, dokud  není kommandem
                 TE
                 smazáno - to zanmená, ùe
pozdëji  spuîtëné  programy  jsou  rovnëù  prováděny  v  interaktivním
monitorovacím módu.

   Podrobnosti nalezeznete v kapitole zabývající se
                 monitorováním
                .
```

1.276 Popis pêíkazu WAITFORPORT (Trace Start)

```
WaitForPort [Jméno portu]

   WaitForPort  çeká  deset  sekund, neù je zadaný port pêipraven.  Vrácená
hodnota  0  znamená - port nalezen.  Vrácená hodnota 5 znamená, ùe aplikace
nebëùí  nebo  port  neexistuje.  U jmen portû je rozliîováno mezi velkými a
malými písmeny.

Pêíklad:
WaitForPort ED_1

   Çekat na spuîtëní portu tímto zpûsobem není zrovna nejvhodnëjîí, protoùe
spolu s programem çeká také uùivatel, kterému by se nemuselo líbit çekat 10
sekund.  Mnohem  lepîi je na spuîtëní portu poçkat aù v samotném programu,
napêíklad takto:

  DO pocet=1 Until Show('P','<název portu>')
    Call Delay(20)
    IF pocet=100 Then DO;SAY 'Zadaný port nebyl nalezen!';Exit 20;End
  End
```

1.277 Informace o pêesmërování výstupu ukázkových programû

```
Poznámka ke spouîtëní programûm

   Pêi spuîtëní jakéhokoliv ukázkového programu si program "Programy.rexx"
```

otevêe výstupní obrazovku. Pokud pro vás bude její velikost nevhodná, mûùete si výstupní obrazovku nadefinovat sami. Stací jen mít v adresáêi "ENV:" soubor se jménem "ARM_OUT" a v nëm napsaný výstup. Napêíklad takto: "CON:10/10/630/246/Super obrazovka/WAIT/CLOSE" atd. (Pozor! Nesmíte udëlat chybu.) Jednosuíe to provedete v Shellu pêíkazem SetEnv ARM_OUT "<výstup>" a soubor pak musíte pêekopírovat do ENVARC:, aby se vám po resetu nesmazal. Lze to provést pêíkazem COPY ENV:ARM_OUT ENVARC:.

Máte vîak taky moùnost zmënit výstup pêímo v programu. Nëkde na zaçátku se totoù nastavuje promënná "Z_STDOUT", která definuje vzhled výstupního okna. Soubor v ENV: vîak má vyîîí prioritu neù tato promënná, tzn., ùe pokud existuje nemají ùádné zmëny této promënné vliv.

Pokud neumíte nastavovat vzhleded a vlastnosti okna a nemáte ùádnou dokumentaci, mohu vám poskytnout çeský návod pro velice zajímavý prográmek, jenù se jmenuje "KingCon". V nëm naleznete návod jak tímto zpûsobem otvírat okna. Vêele vám doporuçuji sehnat si celý program,protoùe jeho sluùby jistë vyuùijete. A jeîtë jedna dobrá zpráva. Tento program je volnë îíêitelný.

Chci se podívat na dokumentaci KingCONu

Program lze stáhnout z Aminetu. Verze 1.3 se nachází jiù na Aminet SETu 2. (util/shell/KingCON_1_3.lha)

1.278 Zkrácené abecední seznamy vîech funkcí a pêíkazû

Pêíkazy:

ADDRESS

ARG

BREAK

CALL

DO

DROP

ECHO

EXIT

IF

INTERPRET

ITERATE

LEAVE

NOP

NUMERIC

OPTIONS

PARSE

PROCEDURE

PULL

PUSH

QUEUE

RETURN

SAY

SELECT

SHELL

SIGNAL

TRACE
   Sekundární pêíkazy: (nemûùou existovat samostatnë)

ELSE

END

OTHERWISE

WHEN
Abecední seznam interních funkcí a funkcí knihovny "rexxsupport. ↩
   library"

ABBREV()

ABS()

ADDLIB()

ADDRESS()

ALLOCMEM()

ARG()

B2C()

BADDR()

BITAND()

BITCHG()

BITCLR()

BITCOMP()

BITOR()

BITSET()

BITTST()

BITXOR()

C2B()

C2D()

C2X()

CENTER()

CLOSE()

CLOSEPORT()

COMPARE()

COMPRESS()

COPIES()

D2C()

D2X()

DATATYPE()

DATE()

DELAY()

DELETE()

DELSTR()

DELWORD()

DIGITS()

EOF()

ERRORTEXT()

EXISTS()

EXPORT()

FIND()

FORBID()

FORM()

FREEMEM()

FREESPACE()

FUZZ()

GETARG()

GETCLIP()

GETPKT()

GETSPACE()

HASH()

IMPORT()

INDEX()

INSERT()

LASTPOS()

LEFT()

LENGTH()

LINES()

MAX()

MIN()

NEXT()

NULL()

OFFSET()

OPEN()

OPENPORT()

OVERLAY()

PERMIT()

POS()

PRAGMA()

RANDOM()

RANDU()

READCH()

READLN()

REMLIB()

REPLY()

REVERSE()

RIGHT()

SEEK()

SETCLIP()

SHOW()

SHOWDIR()

SHOWLIST()

SIGN()

SOURCELINE()

SPACE()

STATEF()

STORAGE()

STRIP()

SUBSTR()

SUBWORD()

SYMBOL()

TIME()

TRACE()

TRANSLATE()

TRIM()

TRUNC()

TYPEPKT()

UPPER()

VALUE()

VERIFY()

WAITPKT()

WORD()

WORDINDEX()

WORDLENGTH()

WORDS()

WRITECH()

WRITELN()

X2C()

X2D()

XRANGE()

1.279 Rychle najdete vîe, co potêebujete

Rejstêík vîeho, co potêebujete rychle najít

Abecední seznamy Základní pojmy

Pêíkazy Arexu

Standartní datové proudy

Integrované funkce

Booleova algebra

Výrazy

Funkce knihovny RexxSupport

Token...

Operátory

Zkrácené Funkce a Pêíkazy

Komand. rozh.

Klausule

Pûvodní a nové programy
Popis Speciality

? Funkce v ARexxu ?

Monitorování

Odchycení chyb

!! Chybová hláîení !!

Syntaxní analýza

Îablony

Pomocné (kommandové) programy

1.280 Seznam vîech programû

Rejstêík vîech ukázkových programû

Vyberte si program jenù chcete spustit:
! Poznámka !
1.
Informace o programu
2.
Informace o programu
3.
Informace o programu
4.
Informace o programu
5.
Informace o programu
6.
Informace o programu
7.
Informace o programu
8.
Informace o programu
9.
Informace o programu
10.
Informace o programu
11.
Informace o programu
12.
Informace o programu
13.
Informace o programu
14.
Informace o programu
15. "Interrupt.rexx" " LINK Pêeruîení 180}@{

```
Doplnëné ukázkové programy:
 1.
                 Násobilka
                 2.
                 Manipulace s êetëzci
                 3.
                 Bitové manipulace
                 4.
                 Adresy
                 5.
                 Barevný pêechod...
                   (Bonusový program, nesouvisí s výukou ARexxu)

Poznámka:
 Vîechny tyto programy jdou spustit díky programu " Programy.rexx ".

 Tento program vîak také mimo jiné umí velice zajímavou vëc.Zavoláte-li jej
 s parametrem  "CUC" pêepne  se d o tzv. "Vycucávacího" reùimu, (Pojmenoval
 autor.) který  vám umoùní  si kterýkoliv program "vycucnout" a  uloùit jej
 do souboru. Ti  zvás, kteêí si program mûùou  jednoduîe  zkopírovat z okna
 pêímo do  clipboardu, tuto  vlastnost programu jistë  neocení, ale snad si
 tento program alespoñ prohlédnou.

 Co program potêebuje?
 1. Program vyùaduje knihovnu "rexxsupport.library" - jistë jí máte.
 2. Program  pouùívá knihovnu "rexxreqtools.library",  ale lze  se  bez  ní
    obejít. Program se pokusí reîit situaci jinak.

    Nyní jsem seznámen se základními informacemi a chci spustit
```

1.281 Standartní datové proudy, související pojmy - popis

```
                Popis datových proudû

 STDERR  "Standart  error  device"  Jedná  se o logický název standartního
         chybového  výstupního  zaêízenení,  na které ARexx posílá chybová
         hláîení. Bëhem
                 Monitorování
                 je STDERR smërováno do monitorova-
         cího  okna  (trace  console).   Çíst z tohoto datového proudu lze
         nejlépe pomoci pêíkazu
                 PARSE
                 s volbou
                 EXTERNAL
                .

  STDIN  "Standart  input  device"  Jedná  se  o logický název standartního
         vstupního  zaêízení.   Nejpohodlnëji z nëj lze çíst pomoci pêíkazu

                 PARSE
                 s volbou
                 PULL
                . Pokud je program  spuîtën napêíklad
         ze Shellu, jsou data naçítána odtud jako vstup s klávesnice.
```

STDOUT "Standard output device" Jedná se o logický název standartního výstupního zaêízení, kam je smëêován veîkerý výstup programu, tedy pêedevîím výstup pêíkazu SAY.
Pokud je program spuîtën ze Shellu, je mu automatický pêedán standartní vstup i výstup, ale mnoho programû, ze kterých lze arexxové programy zpouîïet (napê. Multiview), pêedají jako výstup "NIL:", který data smëruje nëkam do "çerné rokle". Vy tedy musíte STDOUT a popê. také STDIN otevêit uvnitê programu funkcí

```
OPEN
('STDOUT',<výstupní zaùízení>,'W').
```

Datové proudy Výstupní nebo vstupní zaêízení, kam je smërován tok dat.

Logické zaêízení (název) Jedná se název datového proudu,který je pouùíván Arexxem. V arexxovém programu si mûùete otevêít libovolný poçet logických souborû (zaêízení). Slouùí k tomu funkce

```
OPEN()
```

, která otevêe logický soubor daného názvu a pêidëlí mu skuteçný objekt, napê. soubor na disku nebo zaêízení CON:, pro výstup do okna, atd.
Do jednoho logického zaêízení nelze zapisovat a souçasnë çíst. Smër datového toku se totiù urçuje uù pêi spuîtëní.

1.282 Ukázkový program

Ukázkový program

Program PD1. Násobilka.rexx Spusï program !
! Poznámka !

```
/* Násobilka - Tomáî Procházka (24.7.1998) */
DO Kolika=2 To 10
  Say '' ; Say '?[2mNásobilka' kolika '?[0m'
  DO i = 1 to 10
     IF kolika*I > 100 THEN LEAVE Kolika
     Say ' 'I '$\times$' kolika '=' I*Kolika
  END
END
```

Program, vypisuje násobilku 2 aù 10. Jednotlivé çásti vùdy oddëluje nadpisem bílé barvy.

Více viz:

DO

SAY

IF

1.283 Ukázkový program

```
                Ukázkový program

   Následující program je mírnë sloùitëjîí. Demonstruje pouùití více funkcí
a pêíkazû. Snad vám pomûùe pêi psaní vlastních programû.

Upozornëní: Nesmíte zadat datum dêívëjîí neù 1.1.1978

 Program PD2.  Manipulace.rexx        Spusï program !
                ! Poznámka !

                Call
                Tiskni('       *1;2mVítám vás v ukázkovém programu.*0m¶¶',4,20)
 Call Tiskni(' Zkuste se mnou chvíli spolupracovat!  Ano?',2,50)
 Call Tiskni('¶ Zadejte údaje ve formátu: *1m<Jméno> <Pêíjmení>,<datum narození>*0 ↩
    m',1,40)
 /* Text bude smazán a znovu napsán jiný ... */
 Call Tiskni('*16D|               ') ; Call Tiskni('*16D|*1m<den.mësíc.rok>*0m ↩
    ',2,30)
 /* Poçítaç si vyùádá vîechny udáje */
 Call Tiskni('¶ Zadejte své údaje: ')

                PARSE PULL
                J1' 'P1','datum1
 Call Tiskni(' Zadejte údaje svého pêítele: ')
 PARSE PULL J2' 'P2','datum2
 Call Tiskni(' Dëkuji, Zpacovávám údaje') ; Call Tiskni('.......¶',10)
 Call Tiskni(' *2mNejpreve se podívám na vaîe datumy narození...*0m¶',2,10)

 /* Následující çást formatuje datumy na: "RRRRMMDD" */

                DO
                 A=1 TO 2

                INTERPRET
                "PARSE VAR datum"A "den'.'mesic'.'rok"
   den   = Right(den,2,'0')      /* Vytvoêení dvoumístného çíslo 01,atd. */
   mesic = Right(mesic,2,'0')

                IF

                LENGTH
               (rok)=2 THEN Rok='19'rok
   INTERPRET "datum"A "=
                COMPRESS
               (rok||mesic||den)"
   INTERPRET "IF LENGTH(datum"A")>8 Then DO;SAY 'Chybné zadání';EXIT 10;END"
 END

 /* Následující çást ukazuje zpûsob pouùítí funkce DATE()*/
 rozdil =
                DATE
               (INTERNAL,datum1,S) - DATE(INTERNAL,datum2,S)
 IF rozdil>0 THEN Text='mladîí' /* rozhodnutí zda jsi mladîí nebo starîí */
             ELSE Text='starîí'
```

```
rozdil=ABS(rozdil)            /* Odstranëní znaménka "-" funkcí
                ABS()
                */
IF rozdil=1 THEN k='en'       /* nastavení koncovky podle poçtu dnû     */
            ELSE k='ny'
IF rozdil>4 Then k='nû'

IF rozdil=0 THEN Call Tiskni('  To je ale náhoda, vy jste stejnë staêí. !?!?¶ ↩
   ',1,50)
            ELSE Call Tiskni('  Jsi o' Rozdil 'd'k Text' neù tvûj pêítel' J2 P2'. ↩
               ¶',1,40)

/* Následující çást si pohraje se jmény. */
Call Tiskni('¶ Tak jak se vám líbil mûj výkon,',2,20)
Call Tiskni('.....,',10) ; Call Tiskni('jsem naprosto dokonalý.¶',,20)
Call Tiskni(' Takùe, Ahoj' J1 P2 'a pozdravuj' J2 P1'.¶',3,40)
Call Tiskni(' Zní mi to trochu divnë, ne ?',3,40) ; Call Tiskni('... ... NE! ¶¶ ↩
   ',4,40)


                Exit
                0


                Tiskni:

                PROCEDURE
                /* Tato procedûra postupnë vypisuje text na obrazovku */
 PARSE ARG Text,Prodleva,Pauza
 IF Prodleva='' Then Prodleva=1
 IF Pauza='' Then Pauza=0
 DO cyklus=0  Until Length(Text)=0
   PARSE VAR Text String +1 Text
   Select
     When String='¶' Then Say ''
     When String='*' Then Do /* úprava êetëzce pro zmënu atributû */
       PARSE VAR Text kontrola'*'.  /* Vysvëtlení */
       Poz = POS('m',Kontrola) + POS('|',Kontrola) + 1
       PARSE VAR Text atributy =Poz Text
       Atributy=COMPRESS(Atributy,'|')
       Call WriteCH(
                STDOUT
               ,'?['atributy) /* Tohle je znak Esc: "?" */
       ITERATE cyklus
     End
    Otherwise DO
      Call WriteCH(STDOUT,String)
      IF Strip(string)~='' Then Call Delay Prodleva
    END
   End
 END
 Call Delay Pauza

                RETURN
                0

-------------------------------------------------------------------------
```

Nemá smysl vysvëtlovat funkci celého programu, protoùe vlastní analýzou se nauçíte mnohem víc. Jen bych chtël poukázat na nëkteré zajímavé çásti.

1.284 Popis procedûry

```
                Tato procedûra je souçástí programu
                 Manipulace s êetëzci
                .

Procedûra T I S K N I
```

Pokud vás tato procedûra (funkce) zaujala, bude asi vhodné se sní blíùe seznámit a pochopit její funkci. Vysvëtlíme si proto podrobnë kaùdý její êádek.

Procedûra zaçíná vlastním názvem a pêíkazem pro vytvoêení lokálních promënných (nových, platících jen pro procedûrû). Dále se pa k pêíkazem

```
                 PARSE ARG
                 pêevádëjí parametry s nimiù byla procedûra volána.

 Tiskni: PROCEDURE
  PARSE ARG Text,Prodleva,Pauza
```

Tyto êádky nastavují promënné "Prodleva" a "Pauza" v pêípadë,ùe je uùivatel nezadá. Nastaví implicitní hodnoty, uîetêí vám tak práci.

```
  IF Prodleva='' Then Prodleva=1
  IF Pauza='' Then Pauza=0
```

Nyní následuje smyçka, která z êetëzce "Text" postupnë ukrajuje písmenko po písmenku a pêídëluje je êetëzci "String". V îablonë je posun zadán relativnë (viz.

```
                 Popis îablon
                ).
  DO cyklus=0  Until Length(Text)=0

                 PARSE VAR
                 Text String +1 Text
```

Následuje oddíl podmínek (

```
                 SELECT
                 ->
                 WHEN
                 ->
                 OTHERWISE
                 ->
                 END
                . )
    Select
```

V pêípadë nalezení '¶' bude pêidán znak pro posun êádku. (viz.

```
                 SAY
                )
      When String='¶' Then Say ''
```

Pokud bude nalezena hvëzdiçka ('*'), program znovu ukrojí z êetëzce "Text",

tentokrát vîak vëtîí çást. Aù po znak 'm' (musí se dodrùet velikost) nebo '|'. Takto vzniklý êetëzec procedûra upraví a poîle najednou do výstupního okna, kde zpûsobí zmënu vzhledu písma nebo vlastností okna.

```
      When String='*' Then Do
```

Nejprve se oddëlí kontrolní êetëzec.Odëlen bude aù po následující hvëzdiçku (pokud jeîtë nëjaká existuje). Tím se zajistí, ùe se ani jeden z výše uvedených znakû nemûùe v tomto êetëzci vyskytnout dvakrát, ale ani oba souçasnë. Následující êádek potom zpoçítá pozici nëkterého ze znakû a k výsledku pripoçte jedniçku. Ta je pak pouùita v pêíkazu

```
                PARSE VAR
               , jako
absolutní pozice (viz.
                Popis îablon
               ).
        PARSE VAR Text kontrola'*'.
        Poz =
                POS
               ('m',Kontrola) + POS('|',Kontrola) + 1
        PARSE VAR Text atributy =Poz Text
```

Funkce

```
                COMPRESS
```

pak odstraní '|', protoùe to není souçástí speciálního êetëzce pro zmënu atributû písma. Zadán vîak musí bít proto, ùe napêíklad pro posun kurzoru doleva je nutné zadat: "*1D". Aby program bezproblémû rozpoznal konec tohoto êetëzce musí se nakonec pêídat: "*1D|". Tento êetëzec je pak zmënen na: "?[1D" a poslán do výstupního okna (konzoly).

```
        Atributy=COMPRESS(Atributy,'|')
        Call
                WriteCH
               (
                STDOUT
               ,'?['atributy) /* Tohle je znak Esc: "?" */

                ITERATE
                cyklus  /* Ukonçí smyçku "cyklus" */
      End
```

Pokud se v ùetëzci "String" nenachází ùádný speciální znak,je prostë poslán do výstupního zaêízení. Pokud se jedná o jiný znak neù o pouhou mezeru, je volána externí funkce

```
                DELAY()
               , program poçká podle vámi zadané hodnoty.
     Otherwise DO
       Call WriteCH(
                STDOUT
               ,String)
       IF
                Strip
               (string)~='' Call Delay Prodleva
     END
    End
  END
```

A nakonec program opët çeká, mûùete takto vytvoêit zpoùdění pêed dalîím vypisováním textu.

```
  Call Delay Pauza
 RETURN 0
```

Pokud jste pozornë çetly, jistë jiù víte, ùe této procedûêe se dájí poslat aù 3 parametry.První je povinný a musí to být êetëzec s textem,který chcete vytisknout do "
STDOUT
" (výstupní zaêízení, napê. okno). Dalîí dva jiù není nutné zadávat, ale umoùnují nastavit zpoùdění výpisu jednotlivých písmen ("Prodleva") a zpoùdění na konci celé procedûry ("Pauza").

Názorné pouùití této procedûry mûùete vidët v doplñkovém ukázkovém programu
Manipulace s êetëzci
.

1.285 Ukázkový program

Ukázkový program

Neù program spustíte, je nutné si nëco vysvëtlit. Program vás hned na poçátku poùádá o zadání çíselného êetëzce. Êëtezec mûùete zadat v libovolné çíselné soustavë, musíte vîak za tímto êetezcem udëlat çárku a informovat poçítaç jakou soustavu jste pouùili a to zadáním: "B" = binární, "X" = hexadecimální nebo nemusíte zadat vûbec nic = znak (ASCII).
Zkuste napsat napêíklad "A" nebo "00100001,b".
Dalîí prûbëh programu je potom naprosto jásný.

```
Program PD3. Bitová_manipulace.rexx     Spusï program !
                ! Poznámka !
               /* Bitové operace, zmëna stavu bitu - TP */


                Options
                Prompt 'Zadej çíselný êetëzec (Maska: êetëzec,çís. soust. {B|X|  ↩
                   }):'
 Pull  String','soustava         /* Uùivatel zadá jeden nebo dva parametry */

                Interpret
                "Stary='"String"'"soustava /* vytvoêí nový êetëzec ??? */

                IF
                soustava~='B' Then
                Say
                'Vámi zadane çíslo vypadá v binární soustavë takto:' C2B(Stary)


                Do
                A=0 BY 1
  /* Následující zpráva bude vypsána aù po druhém prûchodu smyçkou. */
  IF A>0 Then Say 'Pokud zadáte cokoliv jiného neù çíslo, bude peogram ukonçen.'
  Options Prompt 'Zadej pozici çísla jenù má být zmëneno (max.'Length(C2B(Stary) ↩
     -1)'):'

                Parse Pull
                Bit                    /* Uùivatel zadá hodnotu. */
  /* Pokud bit není çíslo, následující êádek opuîtí program. */
```

```
  IF
                Datatype
               (bit)='CHAR' Then Exit 0
  Novy =
                BITCHG
               (Stary,Bit)
  /* Výpis zprávy, která podrobnì informuje o provedené zmìnì. */
  Say ''
  Say '           Binární podoba   Decimální podoba   Znak tabulky ASCII'
  Say 'Pûvodní: ?[2m '
                Center
               (
                C2B
               (Stary),14)'  'Center(
                C2D
               (Stary),16)'  'Center(Stary,18)'?[0m'
  Say 'Zmìnìný: ?[2m 'Center(C2B(Novy),14)'  'Center(C2D(Novy),16)'  'Center(Novy ↩
     ,18)'?[0m'
  Say ''
END
```

Poznámka: Êádek, "Interpret "Stary='"String"'"soustava" má zajímavou funkci. Vámi zadané parametry totiù interpretuje tak, ùe vytvoêí nový êetezec. Pêíklad: Zadáte-li: "01000001,b" pêíkaz tyto parametry pêeloùí takto: "Stary='01000001'b". Zajímave, ùe ?

Tento program doufám,ùe naprosto jasnì demonstruje pouùití funkce BITCHG(), ale také nìkolika dalîích.

Úçelem programu pûvodnì bylo pochopit princip funkce BITCHG(), ale snad vám pêinese i nové poznatky v programování.

1.286 Ukázkový program

Ukázkový program

Program Adresy.rexx demonstruje, co vîechno lze získat z hlubin pamìti vaîeho poçítaçe.

Neù se vîak pustíme do zkoumání tohoto programu, chtël bych vás na nìco upozornit: Nikdy se nepokouîejte zapisovat do pamìti, která pro vás nebyla vyhrazena! Çastìji se vám vîak mûùe stát, ùe vlivem chyby se zmìní adresa oznaçující pro vás vyhrazemný úsek pamìti. Takových chyb mûùe nastat spousta, dávejte si na nì dobrý pozor, protoùe mûùou zpûsobit pád systému.

A nyní se jiù podíváme na vlastní program:

```
 Program PD4. Adresy.rexx     Spusï program !
                ! Poznámka !
                /* Adresy.rexx aneb Zjiîtìní pouùívaného Fontu - TP */
 gfxbase =
                showlist
               (l,'graphics.library',,a)
```

```
                Call

                forbid()
                FntAddr =
                next
               (gfxbase,154)
DefFont =
                IMPORT
               (next(FntAddr, 10))
FSize =
                c2d
               (IMPORT(offset(FntAddr, 20),2))
Call
                permit ()

                Say
                ’ Vámi aktuálnë pouùívaný font je: "’DefFont’"’
Say ’ Jeho velikost je’ Fsize’.’
```

Princip programu je vpodstatë velice jednoduchý. Program prostë zjistí polohu "graphics.library" v pamëti, pak zakáùe multitasking a naçte ze správného místa poùadovaný údaj.

Funkce

```
                FORBID()
                 a
                PERMIT()
```

jsou pouùity proto, aby bëhem naçítaní nemohla být pamët zmënëna jiným programem. Navíc je tento zpûsob bezpeçnëjîí.

Aby jste vîak mohli obdobné vëci zkouîet sami musíte znát operaçní systém Amigy, k tomu potêebujete "Amiga Rom Kernel manuál". Pokud vîak máte trochu trpëlivosti, mûùete se do prûzkumu pamëti pustit sami, moùná ùe v ní objevíte nëco zajímavého...

1.287 Ukázkový program

Ukázkový program

Následující program jiù pêímo nesouvisí z výukou ARexxu, ale ukazuje, k çemu lze dokonce pouùít okna shellu a mohl by být pro vás inspirací...

```
Program PD4. Barvy.rexx     Spusï program !
                ! Poznámka !
                /* Barevný pêechod s okny Shellu - TP */

MaxCol= 32 /* poçet barev vaîí obrazovky */
cykly = 2  /* poçet cyklû */

 /* poslední dva parametry fungují jen
    pokud máte nainstalován   KingCON    */
param = ’/NOBORDER/NOSIZE/NOGADS/NOMENU’
```

```
                Open
               (outA,'con: 10/25/150/100/A'param)
 Open(outB,'con:160/25/150/100/B'param)
 Open(outC,'con:310/25/150/100/C'param)
 Open(outD,'con:460/25/150/100/D'param)
 Open(outE,'con:610/25/150/100/E'param)

 MaxCol=(MaxCol-1)*cykly

                Do
                bar=0 to MaxCol
   Writech(outA,'?[>'bar'm')

                Writech
               (outB,'?[>'bar+1'm')
   Writech(outC,'?[>'bar+2'm')
   Writech(outD,'?[>'bar+3'm')
   Writech(outE,'?[>'bar+4'm')

                Call

                delay
               (
                Trunc
               ((MaxCol-bar)%4,0))
 End


                Close
               (outE);Close(outD);Close(outC);Close(outB);Close(outA)
```

Myslím, ùe nemá význam sloùitë popisovat tento jednoduchý program program. Prostë se otevêe nëkolik oken a v nich se pak pomoci êídící sekvence (ANSI) mëní postupnë barva.

Poslední êádek není v programu nutný, protoùe okna se uzavêou po skonçení sama, ale pêi spuîtëní z ARexx manuálu by pêekrývaly dotaz na vycucnutí programu a zavêely by se aù pozdëji. A navíc se takto zavírají efektivnëji (v opaçném poêadí).