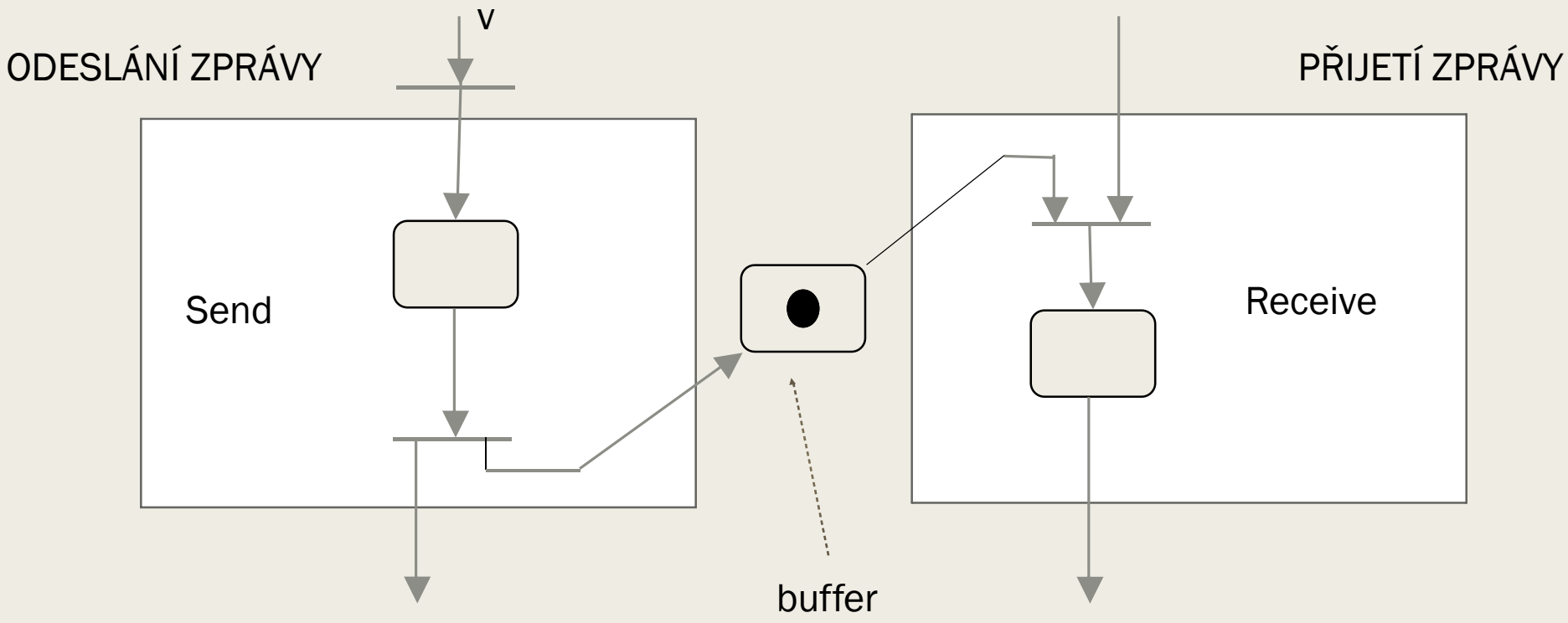
KOMUNIKUJÍCÍ SYSTÉMY

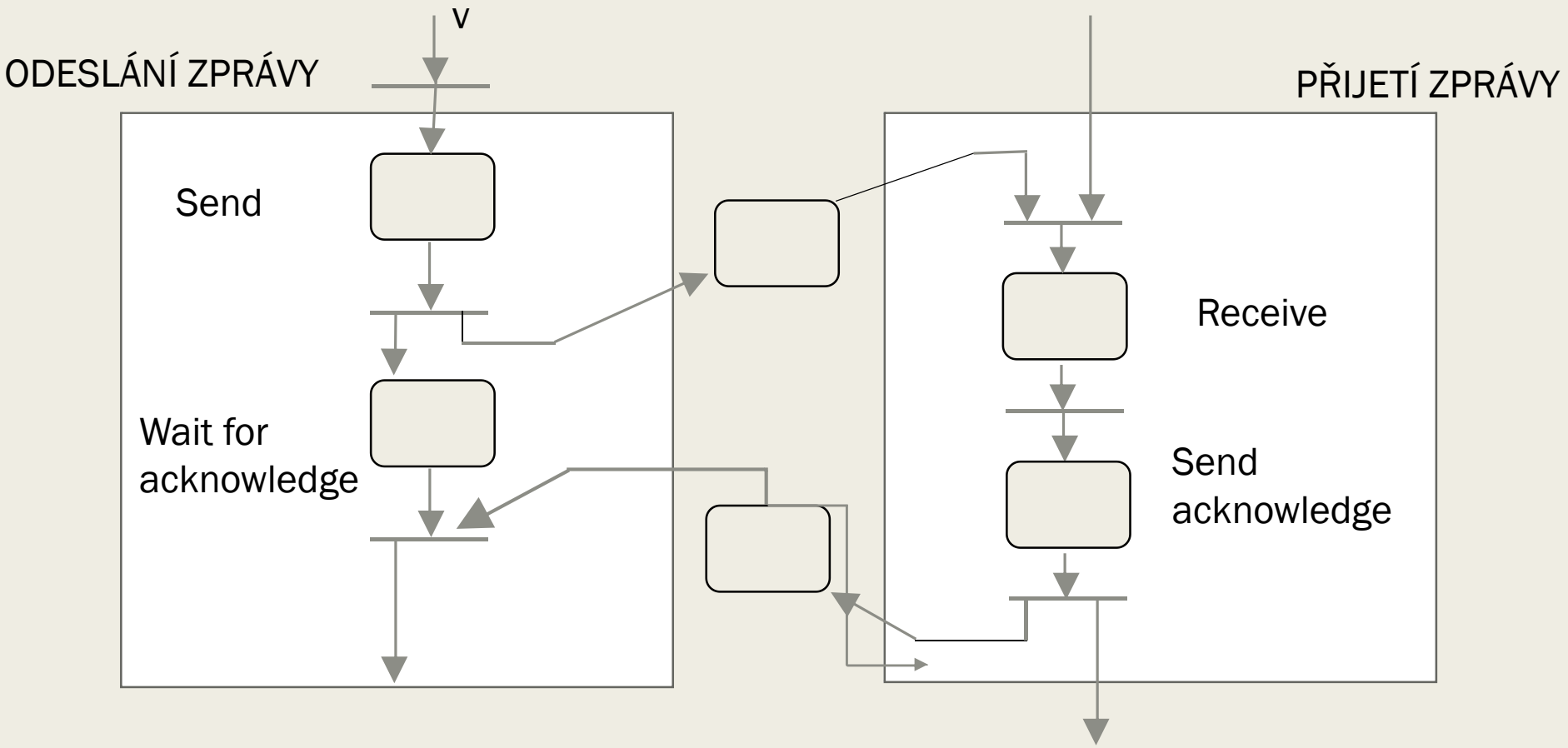
Systémy s předáváním zpráv

- Dvě základní primitiva
  - *send(channel, msg)*
  - *receive(channel, msg)*
- Komunikační kanál, buffering
  - *Asynchronní kanál*
    - neomezená kapacita
    - operace `send` je neblokující
    - složitější implementace
    - příklad: e-mail
  - *Synchronní kanál*
    - omezená kapacita X nulová kapacita
    - `send` může být blokující
    - počet příjemců
      - *Jeden příjemce - kanál*
      - *více příjemců - mailbox*

Asynchronní komunikace



Synchronní komunikace



Simulace synchronního kanálu skrz asycnchronní

- Nulová kapacita
  - `send(ch, msg) ⇒ send(ch1, msg)`
    `receive(ch2, dummy)`
  - `receive(ch, msg) ⇒ receive(ch1, msg)`
    `send(ch2, „ack“)`
- Nenulová kapacita (n zpráv)
  - `init(ch) ⇒ n_times {  send(ch2, „ack“)`
    `....`
    `send(ch2, „ack“)`
    `}`

MODELY KOMUNIKUJÍCÍCH PROCESŮ

CSP, π – kalkul, OCCAM

CSP – Communicating Sequential Processes

- Communicating Sequential Processes (CSP) je formální jazyk vyvinutý pro modelování paralelních systémů komunikujících předáváním zpráv.
- C. A. R. HOARE 1978
- Výchozí pro další systémy, jako jsou napríkad π-kalkul nebo OCCAM

OCCAM

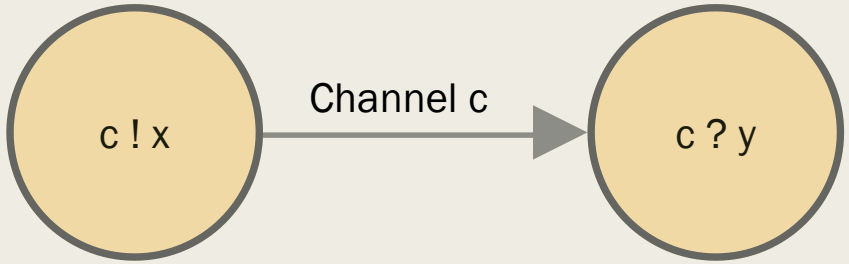
- Původně založen na Hoareho CSP, později inspirován a rozšířen o prvky z PI-kalkulu (OCCAM PI)
- Navržen tak, aby měl formální sémantikou vhodnou pro automatickou transformaci programu
- Transformace programů zapsaných v OCCAMu přímo na hardware (transputery, SW-HW codesign)
- Řada jazyků má k dispozici OCCAM – like rozšíření pro paralelní procesy, včetně JAVA

OCCAM, základní primitiva

- Occam primitivum je *proces* a je jednoho z následujících pěti druhů:

Přiřazení	`x := y + 2`
Vstup	`keyboard ? char`
Výstup	`screen ! char`
Skip	`SKIP – NOP, které se ukončí`
Stop	`STOP -- NOP, které se nikdy neukončí`

- Kanály poskytují možnost komunikace mezi procesy, nebufferovanou point-to-point synchronní komunikaci
- Kanály mají definovaný typ zpráv, které přenášejí



Sekvenční procesy

- SEQ executes sub-processes sequentially

```
SEQ
      keyboard ? char -- read char from keyboard
      screen ! char -- write char to screen
```

Můžeme replikovat sekvence

```
SEQ i = 0 FOR array.size
      stream ! data.array[i]
```

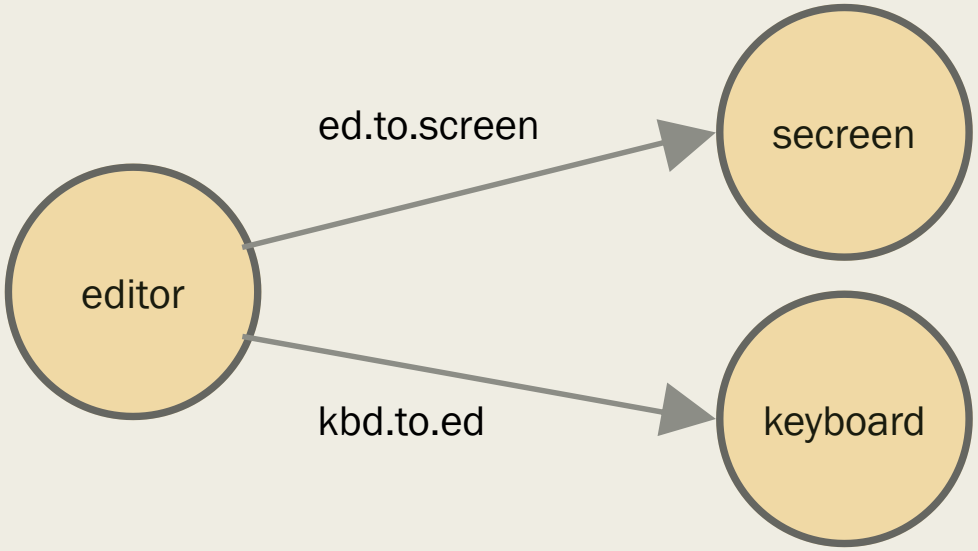
... je stejné jako

```
SEQ
      stream ! data.array[0]
      stream ! data.array[1]
      ...
```

Kompozice paralelních procesů

- PAR spustí vykonávání paralelních procesů

```
PAR
        keyboard(kbd.to.ed)
        editor(kbd.to.ed,ed.to.screen)
        screen(ed.to.screen)
```



PAR pro paralelní vykonávání procesů

- Příklad: Square pipe

```
WHILE next <> EOF
      SEQ
            x := next
            PAR
                  in ? next
                  out ! x * x
```

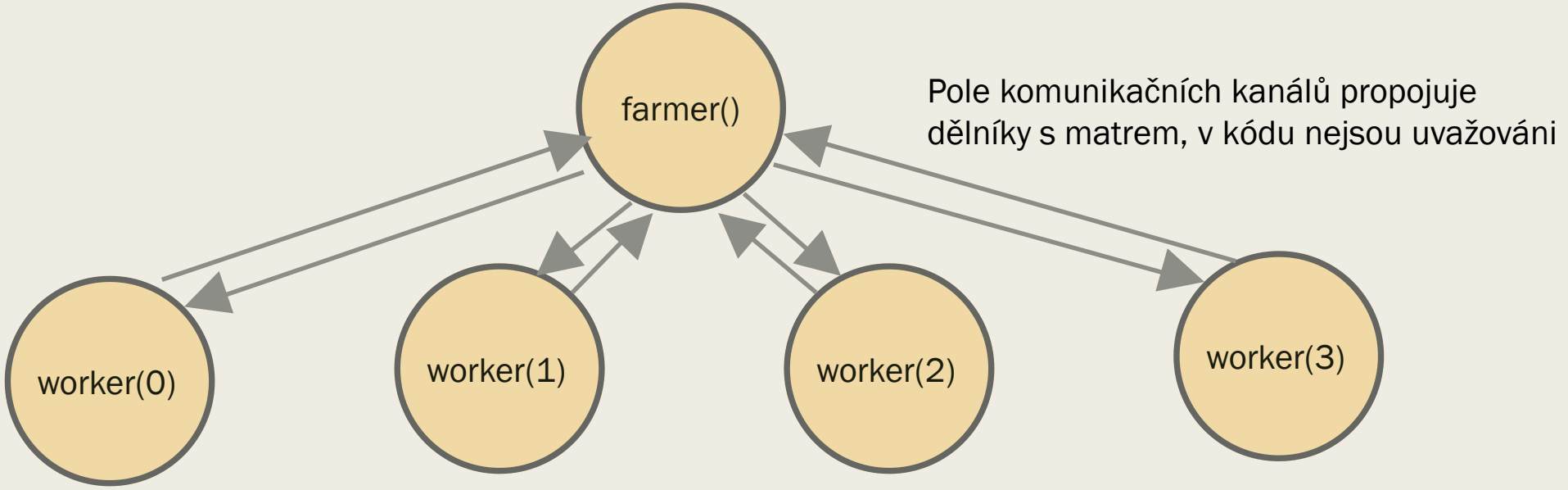
- Omezení přístupu paralelních procesů k datům
  - *Variables modified in one arm of PAR cannot be read or written in other parts of PAR, e.g., PAR -- this PAR is invalid*

```
SEQ
      mice := 42 -- assigns to mice
      c ! 42
      c ? mice -- assigns to mice
```

Replikovaný PAR

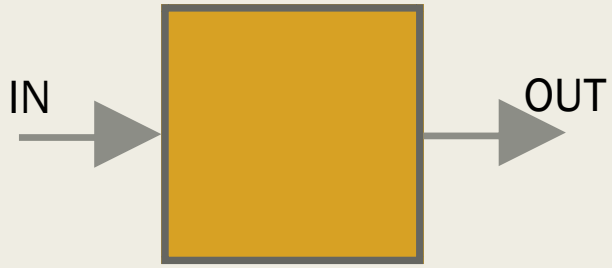
- Replikovaný PAR může být použit pro vytvoření pole paralelních procesů

```
PAR
        farmer()
        PAR i = 0 FOR 4 -- count must be constant
              worker(i)
```



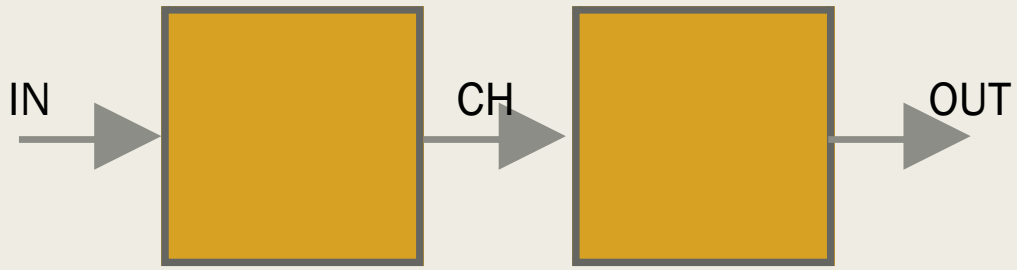
Pole komunikačních kanálů propojuje dělníky s matrem, v kódu nejsou uvažováni

Vyrovnávací paměť (Buffer)



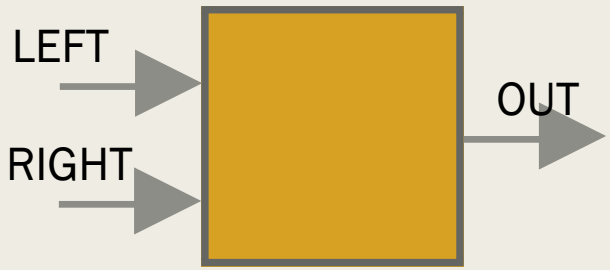

```
WHILE TRUE
    BYTE b:
    SEQ
        in ? b
        out ! b
```

Dvojitý Buffer




```
CHAN OF BYTE ch:
PAR
    WHILE TRUE
        BYTE b:
        SEQ
            in ? b
            ch ! b
    WHILE TRUE
        BYTE b:
        SEQ
            ch ? b
            out ! b
```

Dvojitý Buffer



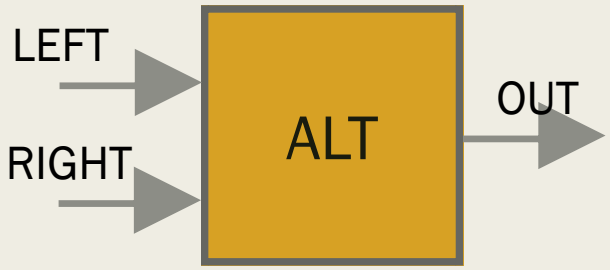

```
WHILE TRUE
    left ? packet
    out ! Packet
    right ? packet
    out ! packet
```

```
WHILE TRUE
    PAR
        left ? packet
            out ! Packet
        right ? packet
            out ! packet
```

Alternace

- ALT combines a number of processes only one of which is executed
- Každý proces má strážce:
  - *vstup input on channel*
  - *čeká na časovač*
  - *Čeká na splnění boolovské podmínky*

```
ALT
        left ? packet -- guard input statement
              out ! packet
        right ? packet -- guard input statement
              out ! packet
```



Alternace

```
ALT
        strazce1
                proces1
…
        strazcen
                proces
```

Strážci mohou čekat na

1, vstup

```
        in ? data
```

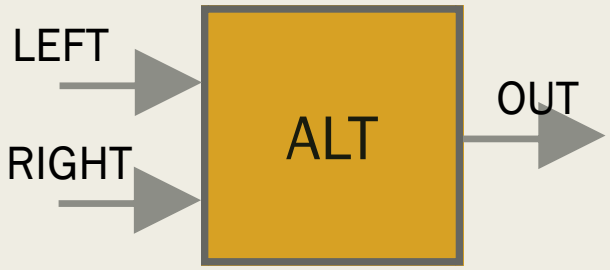
2, platnost boolovské podmínky & vstup

```
        not.empty  &  in ? data
```

2, platnost boolovské podmínky & SKIP

```
        not.empty  &  SKIP
```

```
WHILE TRUE
    ALT
        left ? packet
            out ! packet
        right ? packet
            out ! packet
```



OCCAM – deklarace procedur

```
-- other parts to program
        PROC buff(CHAN OF BYTE in, out)
                WHILE TRUE
                        BYTE x:
                        SEQ
                                in ? x
                                out ! x :
-- end of buff
        CHAN OF BYTE comms, buffer.in, buffer.out:
                PAR
                        buff(buffer.in, comms)
                        buff(comms, buffer.out)
```

π - kalkul

- Formální nástroj pro modelování paralelních a mobilních procesů
- Jazyk + axiomy + redukční pravidla
- Syntax:
  - *Předpona π je v jednom z následujících tvarů* $\pi = \bar{x} < y > \ |x(z) \mid \tau \mid [x=y]$
  - *Proces je pak zapsán jako*

$$P ::= M \mid P|P' \mid \vartheta z P \mid !\, P$$

$$M ::= \ 0 \mid \pi.P \mid M + M'$$

π – kalkul, procesy

$\mathbf{0}$	-	prázdný (ukončený) proces
$\pi.P$	–	předpona
$[x = y]P$	-	test
$P + P'$	-	nedeterministický výběr
$P \mid P'$	-	paralelní kompozice
$(\vartheta z)P$	-	omezení jména
$!\,P$	-	replikace

π – kalkul, redukční pravidla

Je možné přejmenovávat volná jména v procesu, nebo

(INTER) $$\frac{}{\bar{x}\langle z\rangle.P|x(y).Q \rightarrow P|Q\left[\frac{z}{y}\right]}$$

(aplikuje substituci na všechna volná jména v Q)

(R-PAR) $$\frac{P \rightarrow Q}{P|R \rightarrow Q|R}$$

(R-RES) $$\frac{P \rightarrow Q}{(\vartheta x)P \rightarrow (\vartheta x)Q}$$

(R-STRUCT) $$\frac{P \equiv P' \rightarrow Q \equiv Q'}{P \rightarrow Q'}$$

(RESTRUCT) $$\frac{}{\tau.P + M \rightarrow P}$$

π – kalkul, komunikace

- Synchronní přes pojmenovaný komunikační kanál

$$x(y).P|\bar{x}a.P'$$

Tj. Procesy komunikují přes komunikační kanál kanál *x*

$$x(y).P\left|\bar{x}a.P' \rightarrow P\left[\frac{a}{y}\right]\right| P'$$

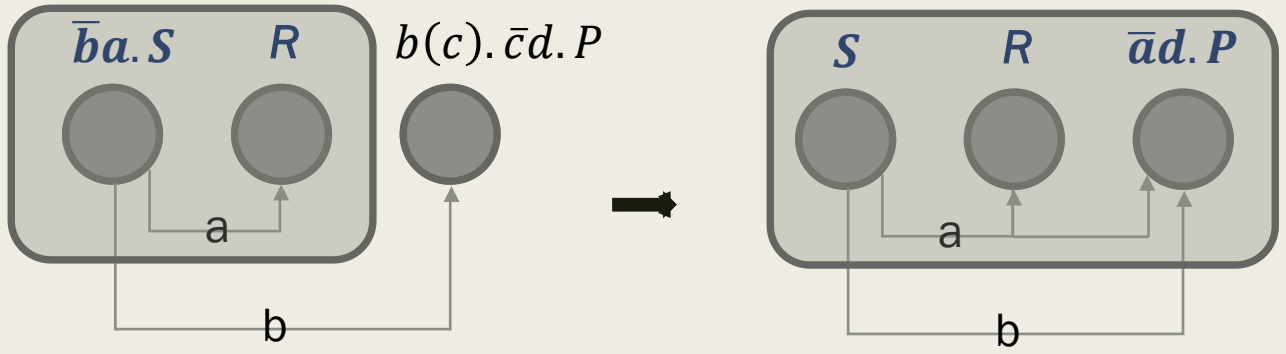
- Jiný příklad ukazuje mobilitu komunikujících kanálů*:*

$$\bar{x}b.\mathbf{0}|x(y).\bar{y}a.\mathbf{0}|b(w).P|c(z).P' \rightarrow$$
$$\mathbf{0}\left|\bar{b}a.\mathbf{0}\right|b(w).P|c(z).P' \equiv$$
$$\bar{b}a.\mathbf{0}|b(w).P|c(z).P' \rightarrow$$
$$\mathbf{0}|P\left[\frac{a}{w}\right]|c(z).p' \equiv P\left[\frac{a}{w}\right]|c(z).p'$$

π – kalkul, omezené komunikační kanály

- Omezení se používá pro soukromé – skryté kanály nějaké skupiny procesů
- Na rozdíl od CSP, kde omezení nemohlo být rozšířeno, v π – kalkulu můžou procesy v dosahuomezení poslat jméno skrytého kanálu vnějšímu procesu a rozšířit tak o něj toto omezení

$$(\vartheta a)(\boldsymbol{\bar{b}a.S}|\boldsymbol{R})|b(c).\bar{c}d.P \rightarrow (\vartheta a)(\boldsymbol{S}|\boldsymbol{R}|\boldsymbol{\bar{a}d.P})$$



π – kalkul, kontext, kongruence

- Kontext zapisujeme s uvedením místa [.]
- Proces *P*, který má být proveden v **kontextu** *C* pak *C*[*P*]
- Procesy jsou v relaci kongruence, pokud jsou v této relaci pro libovolný kontext

$$C_0 : (\vartheta z)([.] \,!\, z(w).\bar{w}a.\mathbf{0})$$
$$C_1 : x(z).\,!\,(\vartheta w)(z(w).[.] + y(v).\mathbf{0})$$

- Aplikací procesu

$$C_0[!\,\bar{z}b.\mathbf{0}] = (\vartheta z)\,(!\,\bar{z}b.0 \mid !\,z(w).\bar{w}a.\mathbf{0}\,)$$
$$C_1[!\,\bar{z}b.\mathbf{0}] = x(z).\,!\,(\vartheta w)(z(w).\,!\,\bar{z}b.\mathbf{0} + y(v).\mathbf{0})$$

- **Kongruence** je relace S, kde $(\mathrm{P}, \mathrm{Q}) \in S \;\Rightarrow (C[P], C[Q]) \in S$ pro jakýkoliv kontext C

Strukturální kongruence

V relaci kongruence jsou procesy, které jsou stejné, akorát jinak zapsány.
V π–kalkulu je strukturální kongruence dána axiomy =>

$M + \mathbf{0} \equiv M$

$\mathrm{M}_1 + \mathrm{M}_2 \equiv \mathrm{M}_2 + \mathrm{M}_1$

$\mathrm{M}_1 + (\mathrm{M}_2 + \mathrm{M}_3) \equiv (\mathrm{M}_1 + \mathrm{M}_2 + \mathrm{M}_3)$

$P|\mathbf{0} \equiv P$

$P_1|P_2 \equiv P_2|P_1$

$(P_1|P_2)|P_3 \equiv P_1|(P_2|P_3)$

$!P \equiv P|!P$

$(\vartheta \mathrm{a})\mathbf{0} \equiv \mathbf{0}$

$(\vartheta a)(\vartheta b)P \equiv (\vartheta b)(\vartheta a)P$

$[\mathrm{x} = \mathrm{x}]\ \pi.\mathrm{P} \equiv \pi.\mathrm{P}$

$\mathrm{P}_1|(\vartheta a)P_2\ pokud\ a \in fn(P_1)$

π – kalkul, akce, tiché (vnitřní) akce

- Provedení akce znázorňujeme štítkovanou přechodovou relací $\rightarrow^{\alpha}$ kde α je akce
- α může být:

xy	výstupní akce
$\bar{x}y$	otevřená výstupní akce
$\bar{x}(z)$	výstupní akce po soukromém kanále
τ	tichá akce

Příklad:

$$\bar{s}a.\bar{s}b.\mathbf{0} \rightarrow^{\bar{s}a} \bar{s}b.\mathbf{0}$$

$$a(y).Q \rightarrow^{a(y)} Q$$

- Tiché akce τ nejsou pozorovatelné (například i komunikace v rámci procesu, nedeterministický výběr apod.)

Příklad:

$$\bar{s}a.\bar{s}b.0|z(y).0|s(v).s(u).\bar{v}u.\mathbf{0} \rightarrow^{\tau} \bar{s}b.0|z(y).0|s(u).\bar{a}u.\mathbf{0}$$

π – kalkul, pozorování

- Pozorování (observables) se vztahují ke komunikačním kanálů, které nejsou soukromé pro nějakou skupinu procesů
- Zápis $\boldsymbol{P \downarrow x}$ – má jméno x jako výsup
- Zápis $\boldsymbol{P \downarrow \overline{x}}$ - má jméno x jako výstup
- Procesy jsou pozorovatelem neroznatelné, pokud mají stejné chování na výstupech
- Příklad:

$$\mathrm{P} = (\vartheta z)(\bar{s}a.\bar{t}b.\mathbf{0}|z(y).\mathbf{0}|s(v).t(u).\bar{v}u.\mathbf{0}|\tau.w(z).\mathbf{0})$$

$\mathrm{P} \downarrow a$??? – ne, je subjekt komunikace, ne komunikační kanál (vstup / výstup)

$\mathrm{P} \downarrow \bar{s}$??? – ano, agent může poslat jméno po kanále s

$\mathrm{P} \downarrow \bar{t}$??? – ne, agent nyní nemůže poslat jméno po kanále t

$\mathrm{P} \downarrow s$??? – ano, agent může přijmout jméno po kanále s

$\mathrm{P} \downarrow t$??? – ne, agent nyní nemůže poslat jméno po kanále t

$\mathrm{P} \downarrow z$??? - ne, jméno z je privátní procesu a není pozorovatelné z vnějšku

$\mathrm{P} \downarrow w$??? – ne, agent nyní nemůže použít w, ale platí $\mathrm{P} \Downarrow w$, viz dále

π – kalkul, podobnost procesů

Centrem zájmu v oblasti procesních algeber je podobnost procesů

Jeden proces může simulovat jiný proces (relace **simulace**)

Bisimulace je relace taková, že procesy se mohou simulovat navzájem

Silná bisimulace ->

Procesy *P* a *Q* jsou v relaci silné bisimulace $S(P,Q)$ pokud

$P \downarrow x$ implikuje Q $\downarrow x$

$P \rightarrow^{\tau} P'$ implikuje $Q \rightarrow^{\tau} Q'$ pro nějaké Q' a $S(P',Q')$

$Q \downarrow x$ implikuje $P \downarrow x$

$Q \rightarrow^{\tau} Q'$ implikuje$P \rightarrow^{\tau} P'$ pro nějaké Q' a $S(P',Q')$

Slabá bisimulace -> tiché akce nejsou pozorovatelné, proto použijeme relaci pro pozorovatelné akce $\Rightarrow^{\alpha}$

Jednou nebo více tichými akcemi přejde proces do stavu dle relace.

$$\Rightarrow \stackrel{\text{def}}{=} (\rightarrow^{\tau})^{*}$$

Pak pro akci α

$$\Rightarrow^{\alpha} \stackrel{\text{def}}{=} \Rightarrow\rightarrow^{\alpha}\Rightarrow$$

Zápis $\boldsymbol{P \Downarrow x}$ značí, že proces P může tichými akcemi přejít ⇒ do stavu, kde $\boldsymbol{P \downarrow x}$

ADA

ADA - programovací jazyk

- Vyvíjen od 80. let jako ‘komplexní’ jazyk poskytující, robustní a obecný nástroj pro progarmování systémů
- Strukturovaný, objektový, výjimky, AGOL + PASCAL + …
- Zahrnující vše, co se zdá být užitečné, ale přesto není stříbrnou kulkou, která vyřeší vše, co je třeba (nejen vlkodlaka)

(viz např zde http://www.cs.unc.edu/techreports/86-020.pdf)

ADA

- **ADA pro synchronizaci používá systém ‘rendezvous’ (aktivní zprávy)**
  - *V Ada je úloha jedním sekvenčním procesem, který má lokální data.*
  - *Úlohy komunikují vyvoláním vstupní procedury jiné úlohy.*
- A má *rendezvous* s těmito úlohami.
  - *Klientská úloha vyvolává vstupní bod který jej může ‘akceptovat’*
- Úloha na straně serveru a způsobí, že mají oba procesy rendevous.
  - *Více akceptovatelných událostí může být na straně serveru očekáváno, pokud je použit příkaz "select";*

ADA - Accept

- **Příkaz accept má tvar**

```
"accept" <entry-name>
[<formal parameter list>]
["do" <statements> "end"];
```

- Příkaz je proveden jen pokud jej vyvolá vzdálený proces

  entry-name (parameters are passed then as well)

  - *After the "end" statement, results are passed back and both tasks are free to continue in parallel*
  - *Either the calling task or the "accept"ing task may be suspendeduntil the other is ready*

ADA – Select, strážci

- **Příkaz Select má následující tvar**

```
– "select"
            ["when" <boolean-expresssion=>]
                  <accept-statement>
            [<statements>]
            {< "or" ["when" <boolean-expression>=>]
                  <accept-statement }
            [<statements>]
            "else" [<statements>]
            "end select;"
```

- 1. Vyhodnotí všechny *boolovské* výrazy, označí všechny “accept” příkazy se splněnou podmínkou jako otevřené *open*. Pokud podmínka není uvedena, pak je *accept* vždy otevřený.
- 2. Zvolte nějaký otevřený "accept" z volaného místa, pokud je otevřených acceptů více, zvolte nějaký nedeterministicky. Pokud žádný accept otevřený není, vykonejte ‘else’ část.
- 3. Pokud tu není žádný "else" a také žádný otevřený "accept„, pak je vyvolána výjimka.

ADA, příklad, omezený buffer

```
task body bounded_buffer is
      buffer: array[0..9] of item;
      in,out: integer;
      count: integer;
      in := 0;
      out := 0;
      count := 0;
      [JÁDRO]
end bounded_buffer;
```

ADA, příklad, omezený buffer [JÁDRO]

```
begin loop
        select
                when count < 10 =>
                        accept insert( it: item )
                                do buffer[in mod 10] := it;
                                        in := in + 1;
                                        count := count - 1;
                                end;
                or when count > 0 =>
                        accept remove( it: out item )
                                do it := buffer[out mod 10];
                                        out := out + 1;
                                        count := count - 1;
                                end;
        end select;
end loop;
```

LINDA

LINDA - úvod

- Vyvinuta AT&T Bell Labs, Yale University (Ahuja, Carriero, Gelenter, LINDA & FRIENDS, 1986)
- Host programming language + LINDA -> Jazyk pro programování paralelních systémů
- Platformě a aplikačně nezávislé
- Původně vyvinuta C-Linda, Fortran-Linda
- Nyní například v Sicstus PROLOG, Agilla (agenti pro WSN)

LINDA

- Paralelní programování založené na jazyce C
- Jedná se o *koordinační jazyk* pro *asynchronní* a *asociativní* komunikační mechanismus nas síleným *globálním prostorem* nazývaným prostor n-tic *Tuples Space* (TS)
- Co je prostor n-tic?
  - *Virtuální sdílená paměť*
- Co je n-tice?
  - *Základní datová struktura nástěnky*
  - *Sekvence aktuálních a formálních položek*
  - *Příklad:*

```
('arraydata', dim1, 13, 2)
```

Nástěnka / prostor n-tic (TUPLESPACE)

Generování n-tic

- **out**
  - *Generuje data (pasivní) n-tice*
  - *Každá položka je vyhodnocena a umístěna na nástěnku*
  - *Řízení je potom navráceno volajícímu procesu*
  - *Příklad:* `out ('array data', dim1, dim2);`
- **eval**
  - *Generuje procesy – aktivní n-tice*
  - *Řízení je navráceno volajícímu procesu ihned po aktivaci nového procesu n-ticí*
  - *Každá položka je teoreticky vyhodnocena souběžně s ostatními a výsledek vložen do n-tice umístěné na nástěnce*
  - *Položky obsahující funkci (nebo podproceduru) způsobí vznik nového procesu, který ji vykoná*
  - Příklad: `eval ("test", f(i));`

Čtení a odstraňování n-tic

- in
  - *Používá šablony pro získání n-tic z nástěnky.*
  - *Pokud je n-tice získána, je odstraněna z nástěnky a dále pak již není zde pro další přístupy.*
  - *Pokud neexistuje n-tice, která by šabloně vyhovovala, je proces pozastaven. Takto lze dosáhnout synchronizace mezi procesy.*
  - *Příklad:* `in("arraydata", ?dim1, ?dim2);`
- rd
  - *Používá šablonu pro získání dat bez jejich odstraňování z nástěnky.*
  - *Po přečtení je n-tice nadále dostupná ostatním procesům.*
  - *Pokud neexistuje n-tice, která by šabloně vyhovovala, je proces pozastaven.*
  - *Příklad:* `rd ("arraydata", ?dim1, ?dim2);`

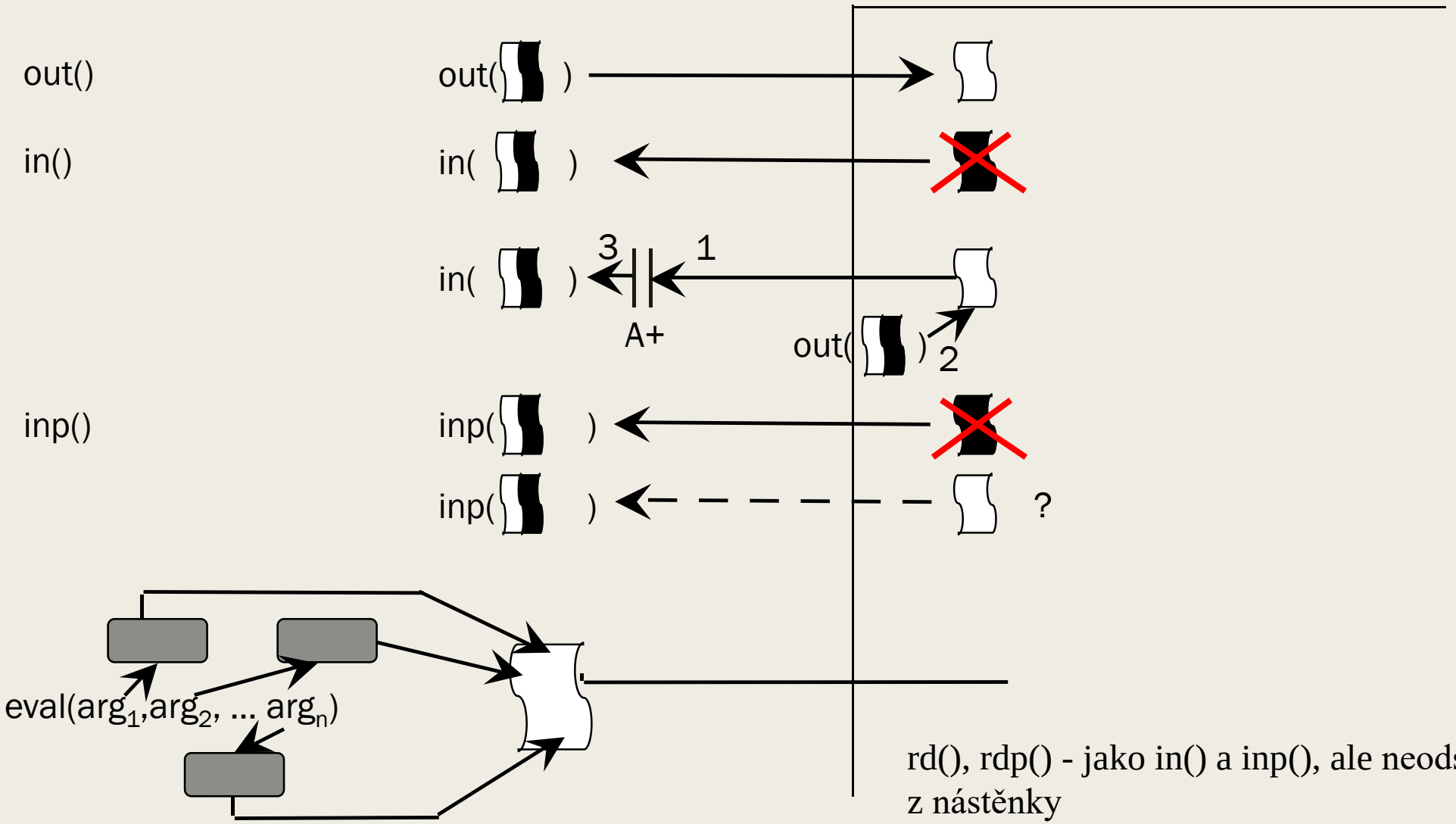
Porovnání n-tice se šablonou

- Musí odpovídat počet položek v šabloně a hledané n-tici
  - *Také musí mýt odpovídající typ, délku, hodnoty*
- Na odpovídající položce musí
  - *Musí odpovídat typ a délka prvků v šabloně s vyhledanou n-ticí*
  - *Pozor – pořadí vyhodnocování položek není definování, proto něčemu jako …*

```
out ("string", i++, i); // bychom se měli vyhnout
```

  - *Pokud je možné šabloně přiřadit více n-tic, není definováno, jak vybrat jednu konkrétní.*

Operátory jazyka LINDA



rd(), rdp() - jako in() a inp(), ale neodstraní tuple z nástěnky

Standardní synchronizační primitiva v LINDA

- *Semafor*
  - P (Sem1) in(Sem1)
  - V (Sem1) out(Sem1)
- *Sdílená paměť*
  - Reading — rd(Variable, ?value)
  - Writing (atomic) — in(Variable, ?value) out(Variable, NewValue)
- *Asynchronní kanál*
  - Send(Chan1, Val) — out(Chan1, val)
  - Receive(Chan1, Val) — in(Chan1, ?val)
  - Ready_to_Receive(Chan1) — rdp(Chan1, ?Dummy)
- *Synchronní kanál*
  - Send(Chan1, Val) — out(Chan1, value, Val) in (Chan1, ?got)
  - Receive(Chan1, Val) — in(Chan1, value, ?Val) out(Chan1, got)

Příklad – pět filosofů

```
phil (i) {
        while (1) {
        think ();
        in(“chopstick”, i);
        in(“chopstick”, (i+1) % Num);
        eat();
        out (“chopstick”, i);
        out (“chopstick”, (i+1) % Num);
        }
}
```

Lístkový algoritmus

```
int incr (name) {
      in (name, ?n);
      out (name, n++);
      return n;
}


{
      ticket = incr (“tick”);
      rd (“next”, ticket);
      ...
      incr (“next”);
}
```

Vyhledání prvočísel

```
is_prime (me) {
limit = sqrt (me) + 1;
for (i=2; i<limit; i++) {
rd (“primes”, i, ?ok);
if (ok && (me % i==0)) return 0;
}
return 1; }
• main () {
for (i=2; i<=LIMIT; i++) eval (“primes”, i, is_prime(i));
for (i=2; i<=LIMIT; i++) {
rd (“primes”, i, ?ok);
if (ok) count++;
}
print (“%d. \n”, count); }
```

Distribuované struktury

- *Seznam* *(Name, Unique_identifier, Next, Val)*
  - *Průchod seznamem*

```
typedef ... TID;
TID id, next;
id = start_of_the_list; /* E.g.. rd(„list“) */
while (id != ID_NIL) {
      rd(„list“, id, ?next, ?val);
      id = next;
}
```

- Vytvoření seznamu

```
TID id, next;
next = ID_NIL;
while (val=getval()) {
      id=GetUniqueId();
      out(„list“, id, next, val);
      next = id;
}
out(„list“, „head“, next)
```

Distribuované struktury

- ***Bag (batoh)***
  - *Nerozlišitelné prvku*
  - *Příklad - master-workers (farm) konstukce používá „bag of tasks“:*
  - *Master Worker*

```
out(„task“, Descript) in(„task“, ?New_Task)
```

- ***Sdílená proměnná***
  - *in(„var1“, ?val)*
  - *out(„var1“, NewVal)*
- ***Pole***
  - *(Jméno, Index/Indicie, Hodnoty)*
  - *Průchod polem*

```
for(i=0; i< MAX, i++)
rd(„Array“, i, ?val);
```

LINDA v SICSTUS PROLOGu

Klient – server přístup

```
| ?- use_module(library('linda/server')).
```

```
| ?- linda.
Server address is msi:’61610’
```

```
| ?- use_module(library('linda/client')).
```

```
| ?- linda_client(msi:’61610’).
```

```
| ?- linda_client(msi:’61610’).
```

```
| ?- linda_client(msi:’61610’).
```

LINDA v SICSTUS PROLOGu

- Neblokující operace `rd_noblock`
- Bag (viz PROLOG) `rd_bag_noblock`
- Na nástěnce: `jmeno(franta), jmeno(pavla), jmeno(hana), jmeno(Jakub)`

```
|?- bagof_rd_noblock(X,jmeno(X),BAG).

BAG = [franta,pavla,hana]
```

LINDA v SICSTUS PROLOGu


```
writepni(L,L).
writepni(A,L):-rd_noblock((prvocislo,A)),write((prvocislo,A)),nl,L2 is
A+1, writepni(L2,L).
writepni(A,L):-L2 is A+1, writepni(L2,L).
writepn(L):-writepni(2,L).

deletepn:-rd_noblock((prvocislo,A)),in((prvocislo,A)),deletepn.
deletepn.

testpni(A,B):- A < B*B, out((prvocislo,A)).
testpni(A,B):- 0=:= (A mod B).
testpni(A,B):- C is B+1, testpni(A,C).

pn(A,A).
pn(A,B):-testpni(A,2),C is A+1, pn(C,B).

do(L):-rd((pnstart,A)),D is A+L, pn(A,D).
```