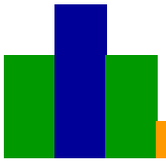
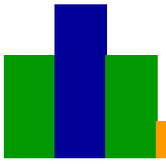


Lecţia 8:

Calcul interactiv: Programe cu registri si voci; Limbajul AGAPIA

v1.0 (24.04.07)
Gheorghe Stefanescu — Universitatea Bucureşti

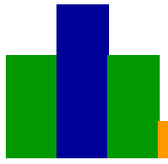
Metode de Dezvoltare Software, Sem.2
Februarie 2007— Iunie 2007



Calcul interactiv

Cuprins:

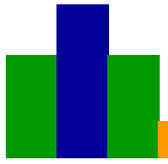
- *Sisteme interactive: Generalitati*
- Specificatii spatio-temporale
- RV-programe (nestructurate)
- RV-programe structurate; AGAPIA v0.1
- Concluzii, diverse, etc.



Sisteme interactive: Generalitati

Sisteme interactive:

- o *clasă importantă* de modele şi sisteme de calcul
- sunt *sisteme deschise*, care interacţionează cu mediul (calculatoare, oameni, mediu fizic)
- încearcă să standardizeze imensa varietate de tehnici de *interacţie a proceselor de calcul* în reţea (fie reţea locală, fie globală - internet)
- încearcă să captureze *interacţia cu mediul nedigital*: interacţia calculatorului cu omul, ori cu mediul înconjurător prin diverşi senzori
- se poate extinde la *sisteme interactive generale* (nedigitale): interacţia celulelor, organismelor, oamenilor, reţelelor naturale ori sociale, etc.

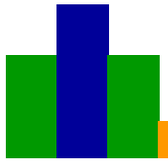


..Sisteme interactive: Generalitati

Vezi recentul compendiu: *Interactive Computation: The New Paradigm*, Ed. D.Goldin, S.Smolka, P. Wagner, Springer, 2006

Probleme specifice:

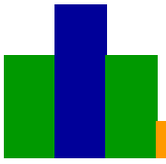
- *Modele pentru interacţie clasice*
  - process algebra, π-calculus, reţele Petri, reţele dataflow
- *Logica interacţiilor*
  - jocuri, logică "liniară", game semantics
- *Componente, interfete, servicii*
  - stream-uri, specificaţii de componente, rafinare
- *Verificarea sistememlor deschise*
  - extensia tehnicii de model checking la sisteme deschise



..Sisteme interactive: Generalitati

Probleme specifice (cont.)

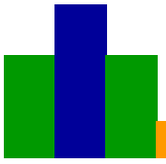
- *Algorithmi online*
  - datele vin online (date partiale); compararea performanţei versiunii online cu cea a versiunii offline
- *Limbaje interactive*
  - sisteme real-time, sisteme îmbarcate (embedded), event-driven programming
- *Human-computer interaction*
  - interacţia om-calculator; delegarea calculatorului pentru job-uri umane; interacţia oamenilor cu ajutorul calculatorului



..Sisteme interactive: Generalitati

Probleme specifice (cont.)

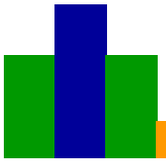
- *Interacţii web*
  - pagini web complexe, e-commerce, agenţi/roboţi, licitaţii electronice, etc.
- *Compunerea interacţiilor; limbaje de coordonare*
  - Linda, modele bazate pe CCS/π-calcul, Reo, etc.
- *De la baze de date la medii de experimentare*
  - medii de date "imperfecte", foarte mari, folosite pentru experimentare, nu doar informare
- *Modelarea şi simularea sistemelor bio/info/socio mari*
  - se pot modela populaţii de indivizi care interacţionează, nu doar simulări statistice



Calcul interactiv

Cuprins:

- Sisteme interactive: Generalitati
- *Specificatii spatio-temporale*
- RV-programe (nestructurate)
- RV-programe structurate; AGAPIA v0.1
- Concluzii, diverse, etc.



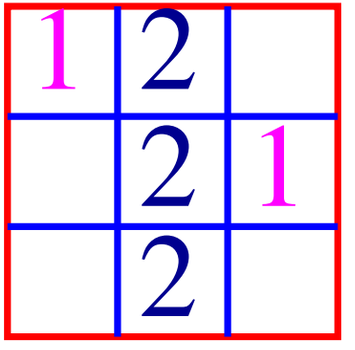
Specificatii spatio-temporale

Vezi Lecţia 3 despre *RV-scenarii* şi:

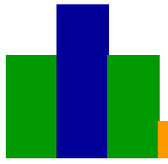
- Tipuri de date temporale
- Specificatii spatio-temporale relationale
- RV-scenarii

In cele ce urmează, analizăm un joc simplu 1-2, pentru a justifica utilitatea folosirii unui *timp local* pe fiecare canal (stream) în locul unui *timp global*.

Jocul 1-2



- Jocul 1-2 este un joc simplu folosit aici pentru a evidenţia *aspectele temporale subtile* ale sistemelor interactive.
- Jocul foloseşte o tablă $n \times n$ şi doi jucători, unul folosind ‘1’, celălalt ‘2’.
- Jucatori alternează la mutare, scriind simbolul lor într-o celulă a tablei.
- Câştigă jucătorul care obţine 3 simboluri consecutive pe linie, coloană, ori diagonală.
- Pentru simplitate, luăm $n = 3$, iar celulele le notăm cu cifre 1-9 (în ordinea stânga-dreapta, apoi sus-jos).



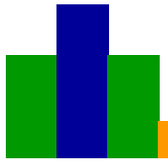
Specificatie

Specificaţia $S : (2,1) \rightarrow (2,1)$ este definită de tuple

$$\langle p0, p1 \mid t \rangle \mapsto \langle a0, a1 \mid t \rangle$$

unde:

- t este registru pentru starea tablei, anume a i-a cifră a lui t descrie conţinutul celulii i. In t codul este:
  – 0 (celulă vidă); 1 (celulă cu 1); 2 (celulă cu 2).
- $p1, p2$ sunt voci de intrare care înregistrează acţiunile jucătorilor iar $a1, a2$ sunt voci de ieşire care stabilesc scorul.
  In $p1, p2$ codul este:
  – 0 (nici o acţiune); $k \in \{1, \ldots, 9\}$ (scrie simbolul în celula k).
  In $a0, a1$ codul este:
  – 0 (mutare greşită/înfrângere); 1 (mutare bună); 2 (victorie).



Exemple

Cîteva exemple de jocuri ('*' înseamnă 'orice'):

(a) un joc complet care se termină cu *remiză*:

$\langle 503040801, 060207090 \mid 000000000 \rangle \mapsto$

$\langle 111111111, 111111111 \mid 121112212 \rangle$

In desen,

1_9	2_4	1_3
1_5	1_1	2_2
2_7	1_7	2_8

(indicii indică ordinea mutărilor).

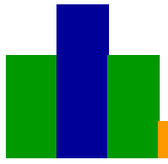
(b) o strategie mai bună pentru jucătorul 1, care *câştigă*:

$\langle 50208****, 06010**** \mid 000000000 \rangle \mapsto$

$\langle 11112****, 11110**** \mid 21**12*1* \rangle$

In desen,

2_4	1_3	
	1_1	2_2
	1_5	



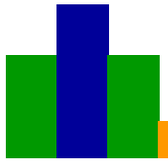
..Exemple

(c) jucătorul *2 nu face o mutare* când este rândul său, deci pierde

$$\langle 5030*****, 0600***** \mid 000000000\rangle \mapsto$$
$$\langle 1112*****, 1110***** \mid **1*12***\rangle$$

(d) în acest caz *ambii jucători scriu* în acelaşi timp, deci jucătorul 2 pierde, căci nu era rândul său

$$\langle 503******, 061****** \mid 000000000\rangle \mapsto$$
$$\langle 112******, 110****** \mid **1*12***\rangle$$



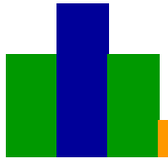
Expresivitatea specificatiilor

Concluzii:

- Este clar că formalismul de mai sus e *suficient de puternic spre a permite o descriere completă a jocului*.
- Mai mult, se pot descrie chiar situaţii când *ambii mută în acelaşi timp*, ori unul *uită să mute*, etc.

Problemă:

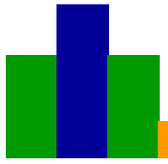
- Rămâne de văzut în ce măsură astfel de specificaţii sunt *naturale* şi dacă ele pot fi *implementate* pe maşini reale, cu constrîngeri spaţio-temporale date.



Timp global vs. timp local

- Vocile se derulează în timp şi ar trebui planificate pe un şuvoi.
- Pentru jocul 1-2 putem găsi ceva convenabil intuiţiei, spre exemplu reprezentând $(503040801, 060207090)$ prin $5^\frown0^\frown0^\frown6^\frown3^\frown0^\frown0^\frown2^\frown4^\frown0^\frown0^\frown7^\frown8^\frown0^\frown0^\frown9^\frown1^\frown0$.
- Să notăm că
  - vocile de intrare se planifică pe şuvoiul de intrare, iar cele de ieşire, pe cel de ieşire, şi
  - într-un joc real, se presupune că jucătorii văd mutările (rezultatul parţial) *"înainte"* de următoarea mutare,

  deci ar trebui să existe o relaţie între *timpul fizic pe şuvoiul de intrare şi cel de pe şuvoiul de ieşire*

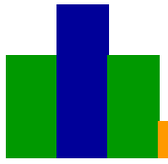


..Timp global vs. timp local

Argument 1. O lecţie de la maşinile cu regiştri:

Registri intrare/iesire diferiti: Regula în cazul maşinilor cu regiştri este să distingem clar între regiştri de intrare şi de ieşire (şi cei locali) spre a putea avea specificaţii mai clare, modulare; cum ei sunt diferiţi, poziţia lor pe bandă e irelevantă;

Voci intrare/iesire diferite: Spre a avea specificaţii spatio-temporale clare şi modulare, ar trebui să urmăm probabil aceeaşi procedură de a diferenţia între vocile/regiştri de intrare şi cei de ieşire.



..Timp global vs. timp local

Argument 2. Jucătorul orb:

Includerea jucătorului: Am spus că în situaţii reale jucătorii trebuie să *vadă* rezultatele *înainte* de noua mutare; asta induce supoziţii despre jucatori (capacitatea lor vizuală, ori mentală), ceea ce nu vrem să introducem în specificaţia jocului.

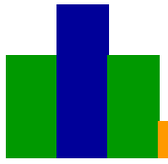
Jucătorul orb: Putem folosi un argument similar cu testul Turing (despre inteligenţa artificială vs. cea naturală):

– să zicem că avem un jucător orb, care *"din pură întâmplare"* alege să joace aceleaşi mutări ca un jucător care vede;

– din punctul de vedere al jocului, nu există nici o diferenţă câtă vreme ei fac aceleaşi mutări;

– în fine, pentru jucătorul orb, nu e nici o diferenţă dacă vede rezultatul după fiecare mutare ori nu.

Concluzie: Este preferabil să avem timpi locali, nu un timp global.



Constrângeri temporale relaxate

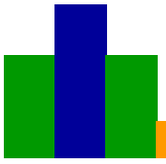
In concluzie, este de dorit să relaxăm condiţia de a avea o relaţie clară între timpul fizic pe şuvoiul de intrare şi cel de ieşire. Atunci:

Teoremă. Cu viziunea de timp local, dacă există suficientă memorie spaţială, atunci orice relaţie calculabilă între vocile de intrare şi de ieşire poate fi implementată.

Dem. Dacă există suficientă memorie staţială, atunci:

– memorăm în regiştri conţinutul vocilor de intrare;
– executăm un program obişnuit (sunt universale);
– livrăm rezultatul în timp vocilor de ieşire. □

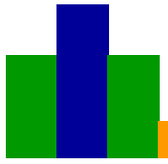
Notă: Presupunerea de *a avea suficient spatiu de memorie* poate fi practic greu de îndeplinit şi atunci trebuie o specificaţie şi un program mai detaliat care să ia în calcul aceste restricţii. Dar acestea vor fi probleme ale altui nivel, cel de implementare în cod maşină.



Calcul interactiv

Cuprins:

- Sisteme interactive: Generalitati
- Specificatii spatio-temporale
- *RV-programe (nestructurate)*
- RV-programe structurate; AGAPIA v0.1
- Concluzii, diverse, etc.



Sisteme interactive

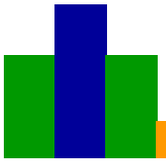
Sisteme interactive:

Sunt multe modele de sisteme interactive

- vexi recentul volum editat de Goldin, Smolka, Wagner

Aici folosim *rv-sisteme* (*sisteme interactive cu regiştri şi voci* [Stefanescu, 2004]; Modelul:

- (1) include *maşinile cu regiştri*, (2) este *invariant la dualitatea spaţiu-timp*, (3) este *compoziţional*, (4) poate descrie *calcule care se extind atât în timp, cât şi în spaţiu*, şi (5) se poate aplica la *sisteme deschise, interactive*.
- Pentru modularitatea în spaţiu modelul foloseşte *voci* (o voce este dualul temporal al unui registru)
- Vocile permit o *organizare a datelor temporale la nivel înalt* - folosite pentru a descrie interfeţele temporale ale proceselor.



..Sisteme interactive

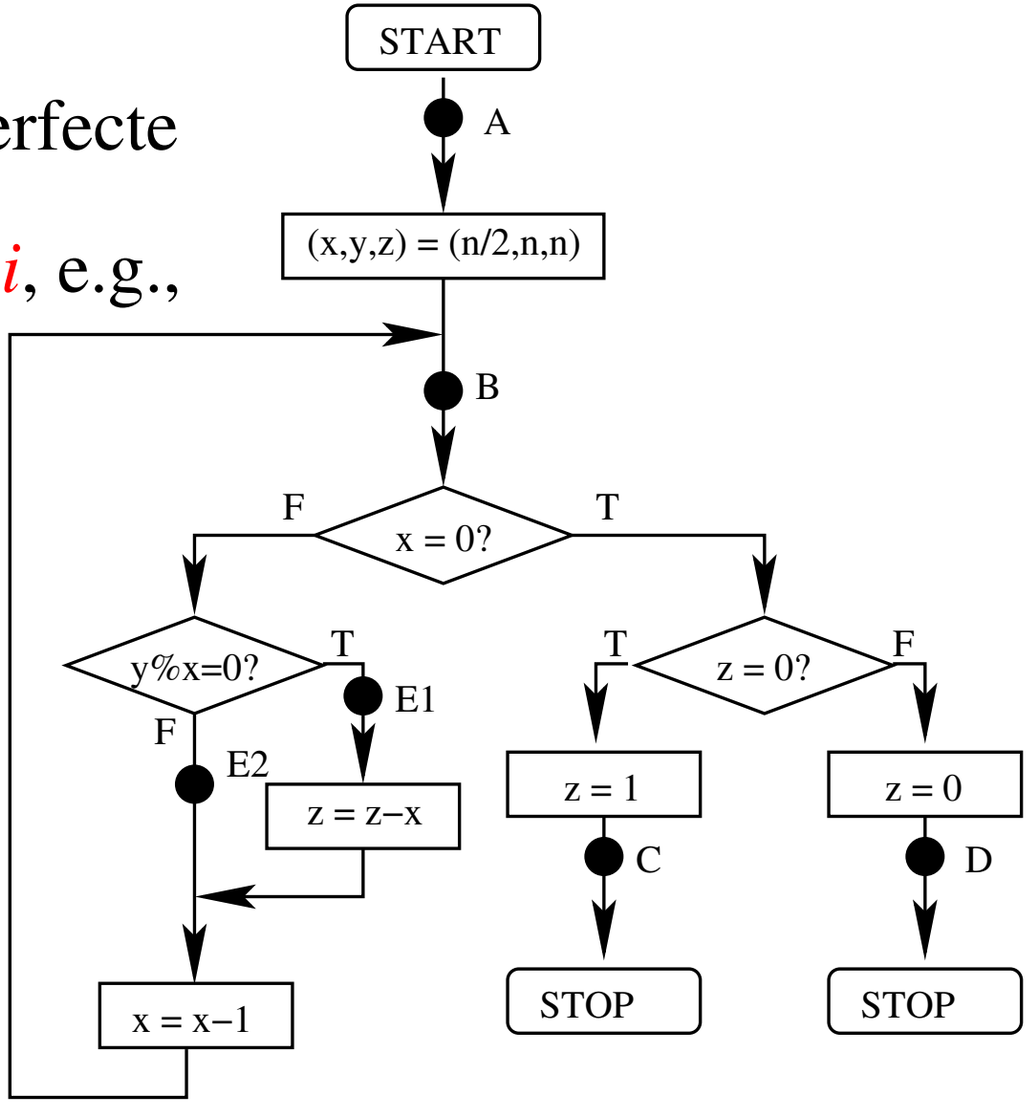
(..Sisteme interactive)

- Ce rezultă este un *limbaj de programare* ce foloseşte tehnici noi pentru sintactică şi semantică (pentru a permite capturarea paradigmei de calcul în spaţiu)
- El permite scrierea de *rv-programe* folosind pentru sintaxă şi pentru semantica operaţională FIS-*uri (sisteme interactive finite)* şi lumbajele lor de griduri.
- Pentru *specificarea* sistemelor interactive se folosesc relaţii între vocile şi regiştrii de intrare şi vocile şi regiştrii de ieşire.

Programe tip schema-logica

Programe tip schema-logica:

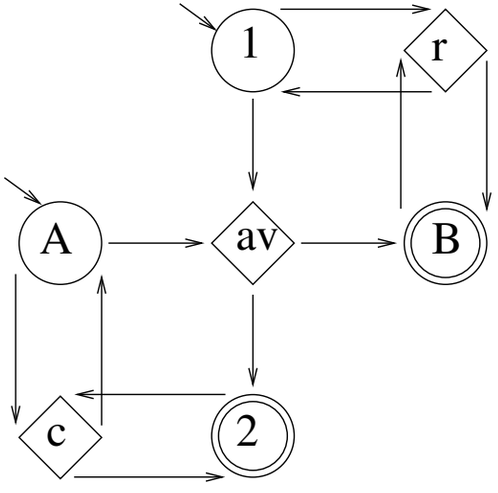
- un program pentru numere perfecte
- *puncte de tăietură* şi *aserţiuni*, e.g.,
  ϕ_B : "$0 \le x \wedge y = n \ge 2$
  $\wedge z = n - \sum_{d|n, x<d<n} d$"
- *condiţii invariante*, e.g.,
  $\phi_B \wedge C_{p(B,E1,B)} \Rightarrow \sigma_2(\phi_B)$
- *terminare*: nu există calcule infinite



Sisteme interactive finite

Sisteme interactive finite:

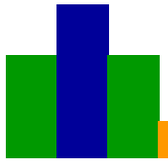
- *stări*: 1,2 [1-iniţială; 2-finală]
- *clase*: A, B [A-iniţială; B-finală]
- *tranziţii*: a, b, c



Procedura de parsare (pentru a accepta griduri):

Parsare pentru
```
abb
cab
cca
```
:

```
 1 1 1     1 1 1     1 1 1     1 1 1          1 1 1
Aa b b    AaBb b    AaBbBb    AaBbBb   ...   AaBbBbB
           2         2 1       2 1             2 1 1
Ac a b    Ac a b    Ac a b    AcAa b          AcAaBbB
                                2               2 2 1
Ac c a    Ac c a    Ac c a    Ac c a          AcAcAaB
                                                2 2 2
```



Specificaţii spatio-temporale

Voci:

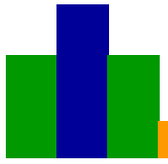
- *stream*: $a_0 \frown a_1 \frown a_2 \frown \ldots$
- *voce* (= *duala temporală a unui registru*):
  - structură temporală pentru a reţine *numere naturale*
  - poate fi utilizată (“auzită”) *în diverse locaţii*
  - în fiecare locaţie o voce specifică o *valoare particulară*
- o voce poate fi *implementată pe un stream*

Specificaţii spatio-temporale relaţionale:

- O *specificaţie spatio-temporală* este o relaţie

$$S \subseteq (\mathbb{N}^m \times \mathbb{N}^p) \times (\mathbb{N}^n \times \mathbb{N}^q)$$

  între voci şi regiştri de intrare şi de ieşire.



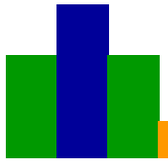
..Specificaţii spatio-temporale

O specificaţie pentru numere perfecte:

Avem trei componente C_x, C_y, C_z unde:

- C_x: "citeşte" n din *nord* şi "scrie" $n^\frown\lfloor n/2\rfloor^\frown(\lfloor n/2\rfloor-1)^\frown\ldots^\frown 2^\frown 1$ în *est*;
- C_y: "citeşte" $n^\frown\lfloor n/2\rfloor^\frown(\lfloor n/2\rfloor-1)^\frown\ldots^\frown 2^\frown 1$ din *vest* şi "scrie" $n^\frown\phi(\lfloor n/2\rfloor)^\frown\ldots^\frown\phi(2)^\frown\phi(1)$ în *est* [$\phi(k) =$ "if k divides n then k else 0"];
- C_z: "citeşte" $n^\frown\phi(\lfloor n/2\rfloor)^\frown\ldots^\frown\phi(2)^\frown\phi(1)$ din *vest*, *scade din primul număr pe celelalte*, şi "scrie" în *sud* "*if the difference is* 0 *then 1 (*`true`*) else 0 (*`false`*)*".

Specificaţia globală in-out $C_x \triangleright C_y \triangleright C_z$: *dacă pe poziţia stângă din nord avem n, atunci pe poziţia dreaptă din sud avem 0 ori 1, fiind* 1 *dnd n este perfect*.



RV-programe

RV-sisteme:

- Un *rv-sistem* (*sistem interactiv cu regiştri şi voci*) este un FIS îmbogăţit cu:
  - *regiştri* asociaţi *stărilor* şi *voci* asociate *claselor*;
  - *transformări spaţio-temporale pentru acţiuni*.

  Studiem rv-sistemele specificate de *rv-programe* (vezi mai jos)
- Un *calcul (run)* este descris de un *rv-scenariu* (un scenariu ca la un FIS, dar cu date concrete în jurul fiecărei acţiuni).

..RV-programe

Un rv-program `Perfect` (pentru numere perfecte):

in: A,1; out: D,2

X::

(A,1)	x : sInt
	tx : tInt;
	tx = x;
	x = x/2;
	goto [B,3];

Y::

(B,1)	y : sInt
tx : tInt	y = tx; goto [C,2];

Z::

(C,1)	z : sInt
tx : tInt	z = tx; goto [D,2];

U::

(A,3)	x : sInt
	tx : tInt;
	tx = x;
	x = x – 1;
	if (x > 0) goto [B,3]
	else goto [B,2];

V::

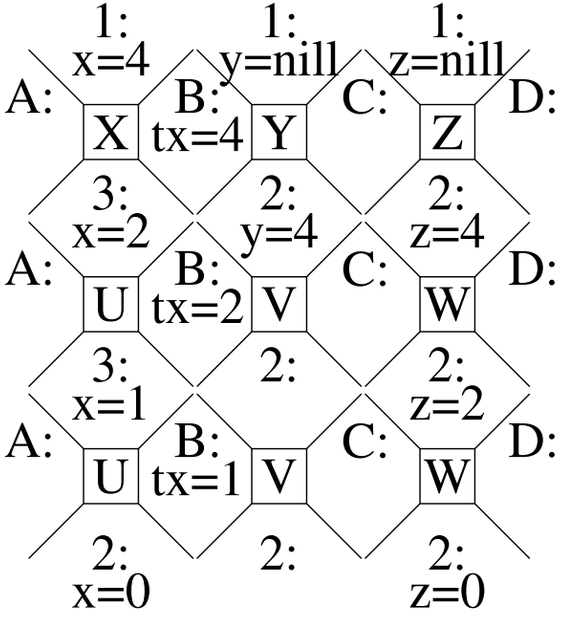
(B,2)	y : sInt
tx : tInt	if(y%tx != 0) tx = 0; goto [C,2];

W::

(C,2)	z : sInt
tx : tInt	z = z – tx;
	if(tx == 1){
	if(z == 0) z = 1
	else z = 0; };
	goto [D,2];

..RV-programe

Scenariu:

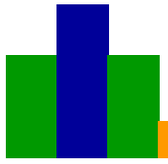


Semantica operationala:

- definită prin scenariile asociate

Semantica relationala:

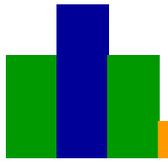
- relaţia in-out generată de scenariile asociate



Programe cu registri

Programele cu regiştri pot calcula *toate funcţiile calculabile* cu:

- *instrucţiuni de bază* simple (*"plus 1", "minus 1", "test la 0"*):
  - (i) `x=x+1` - adună 1 la registrul `x`;
  - (ii) `x=x-1` - scade 1 din registrul `x` (dacă era 0, rămâne 0);
  - (iii) `if x==0 goto e1 else goto e2` - dacă registrul `x` este 0 se trece la instrucţiunea cu eticheta `e1`, altfel la cea cu eticheta `e2`
- *structuri de control* elementare formate din *instrucţiuni cu* `goto` *etichetate*:
  - `e1: x=x+1 goto e2`
  - `e1: x=x-1 goto e2`
  - `e1: if x==0 goto e2 else goto e3`
- *instrucţiuni de intrare-ieşire*



Programe cu registri si voci

Cadru teoretic minimal: Un *rv-program* (ori *program cu regiştri şi voci*) constă din:

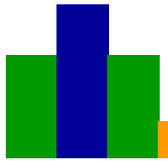
- o *mulţime finită de regiştri şi voci*;
- o *listă de instrucţiuni de tipul*

(i) `(A,a): x = x+1 goto [B,b]`

(ii)
```
(A,a): if x == 0 goto [B,b]
                 else (x = x-1 goto [C,c])
```

unde `x` este registru ori voce şi `A,B,C,a,b,c` sunt etichete

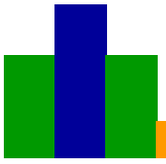
(Definiţia este bazată pe o prezentare echivalentă a maşinilor cu regiştri în care se combină instrucţiunile “minus 1” şi “test la 0”.)



..Programe cu registri si voci

Cadru practic: In practică,

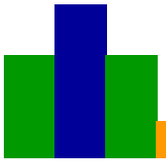
- într-un rv-program vom folosi un *limbaj de programare ceva mai avansat* precum nucleul elementar de programare din limbajele imperative (Pascal, C, Java);
- *sintaxa şi semantica* limbajului se bazează pe FIS-uri şi pe scenariile asociate
- vom prezenta sintaxa şi semantica mai jos;



Sintaxa

Sintaxa:

- Sintaxa este asemănătoare cu cea utilizată în limbajele imperative uzuale.
- Blocul de bază într-un rv-program este *modulul*.
- Un rv-program este o listă de astfel de module.
- Explicăm sintaxa folosind primul modul din programul `Perfect`.

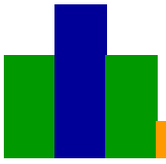


..Sintaxa

Primul modul are un nume `X` şi patru zone:

Zona stânga-sus:

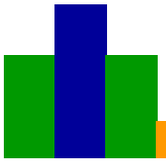
- aici avem o pereche de etichete `(A,1)` care specifică coordonatele de interacţie şi control necesare pentru a aplica modulul;
- să notăm că perechi similare apar în zona de cod din dreapta-jos, dar formatul `[B,3]` şi sensul sunt diferite.



..Sintaxa

Zona dreapta-sus:

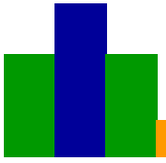
- aici se declară variabilele de intrare spaţiale;
- aceste variabile specifică starea memoriei înainte de aplicarea modulului;
- pentru a le distinge de celelalte variabile, prefixăm tipul lor cu “`s`”;
- modulul `X` are o variabilă de intrare de tip întreg spaţial, notat `sInt`.



..Sintaxa

Zona stânga-jos:

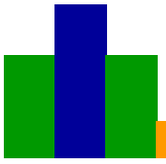
- este similară cu partea dreapta-sus, dar declară variabilele temporale de intrare;
- astfel de variabile apar la interfaţa de interacţie a modulelor;
- prefixăm tipul acestor variabile ce "`t`" (datele au reprezentare temporală)
- modulul `X` nu are variabile temporale.



..Sintaxa

Zona stânga-jos:

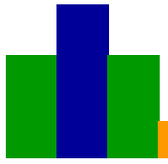
- această zonă conţine corpul modulului, incluzând declaraţii locale de variabile şi cod, simular cu codul C;
- în modul nu se face distinţie între tipul spaţial or temporal al variabilelor, codul utilizîndu-le liber
- calculul într-un sistem interactiv are două dimensiuni – una verticală (capturînd secvenţa de calcul a firului de execuţie), cealaltă orizontală (care specifică o secvenţă de interacţie între modulele)
- ieşirea din modul se face cu o instrucţiune `goto`;



..Sintaxa

- o instrucţiune ca `goto [B,3]` specifică următoarele:
  - ∗ datele variabilelor spaţiale din modulul curent se folosesc într-un nou modul cu eticheta de control `3`;
  - ∗ ne se ştie din modulul curent care vor fi datele de interacţie în noul modul (ele se află folosind proprietăţile generale ale sistemului interactiv)

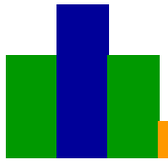
  şi similar
  - ∗ datele variabilelor temporale din modulul curent se folosesc la interfaţa de interacţie a unui nou modul cu eticheta de interacţie `B`;
  - ∗ nu se ştie din modulul curent care va fi starea memoriei acolo, depinzând de proprietăţile generale ale sistemului interactiv



Semantica operationala

Semantica operationala: Explicaţii privind construcţia scenariilor (folosind exemplul de la programul `Perfect`):

- Fiecare celulă din grid are o etichetă din {`X`,`Y`,`Z`,`U`,`V`,`W`} care indică modulul din program folosit în acea celulă; fiecare celulă are date concrete pentru stări în zonele de sus/jos şi date concrete pentru interacţie în zonele stânga-dreapta; gridul se construieşte din stânga-sus în direcţia dreapta-jos completând noi celule când datele din stânga-sus sunt deja calculate.
- O zonă poate conţine informaţii adiţionale ca `y=4` indicând că în acea zonă valuarea lui `y` devine `4`.

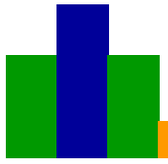


..Semantica operationala

..(scenarii, cont.)

- Informaţia completă despre starea curentă a unui proces [celule] se obţine colectând pe verticală variabilele spaţiale cu ultimile lor valori actualizate. Exemplu: `V` din coloana a doua, jos are `y=4`. Similar, informaţia completă privind variabilele temporale se obţine colectând ultimile valori mergând orizontal spre stânga. Exemplu: `W` din linia a 2-a are `tx=2`.

- Prima coloană are o clasă de intrare (aici `A`) şi tuple particulare de valori pentru variabilele sale temporale. Prima linie are o stare iniţială (aici `1`) şi tuple particulare de valori pentru variabilele sale spaţiale.



..Semantica operationala

..(scenarii, cont.)

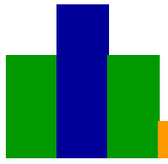
- Cu cele de mai sus, calculul local al unei celule α constă în:

  (i) Se ia un modul β al programului care are ca etichetă de (clasă,stare) ce este în zonele (stânga,sus) ale celulii α.

  (ii) Se execută codul din β folosind valorile concrete din α (stânga,sus) pentru variabilele temporale şi spaţiale.

  (iii) Dacă execuţia locală în β se termină cu o instrucţiune `goto` $[\Gamma, \gamma]$, etichetăm zonele dreapta (resp. jos) ale lui of α cu Γ (resp. γ).

  (iv) Inserăm în zonele dreapta (resp. jos) ale lui of α valorile variabilelor temporale (resp. spaţiale), actualizate de β.

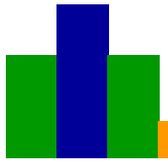


..Semantica operationala

..(scenarii, cont.)

- Un *scenariu parţial* (pentru un rv-program) este unul construit cu regulile de mai sus.
- Un scenariu este *scenariu complet* dacă în dreapta sunt numai clase finale, iar jos numai stări finale.

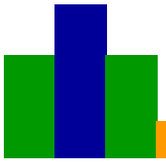
Exemplul dat este un scenariu complet pentru programul `Perfect`.



Calcul interactiv

Cuprins:

- Sisteme interactive: Generalitati
- Specificatii spatio-temporale
- RV-programe (nestructurate)
- *RV-programe structurate; AGAPIA v0.1*
- Concluzii, diverse, etc.

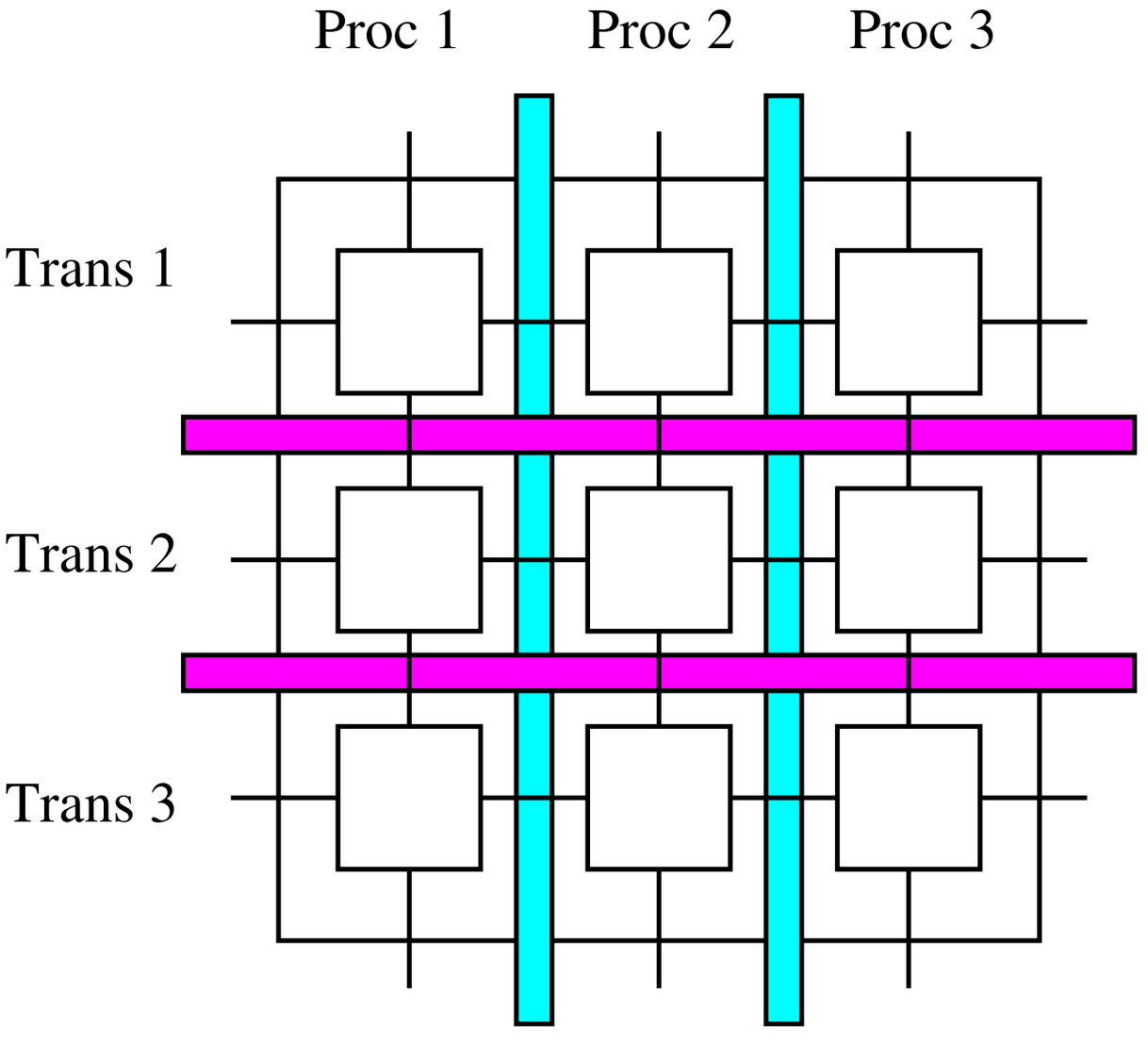


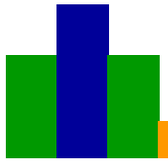
AGAPIA

Lumbajul AGAPIA: caracteristici de bază

- este *invariant la dualitatea spaţiu-timp*
- conţine structuri de *date temporale de nivel înalt*
- *calculul se extinse* atât în *timp*, cât şi în *spaţiu*
- este model *compoziţional*
- permite *programare structurată* (funcţională şi interactivă)
- are o *semantică operaţională* simplă (folosind *scenarii*)
- are o *semantică relatională* simplă

Procese si tranzactii





AGAPIA v0.1: Sintaxa

Sintaxa limbajului AGAPIA v0.1:

Interfete

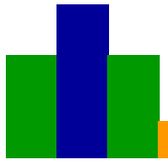
$SST ::= nill \mid sn \mid sb$
$\quad \mid (SST \cup SST) \mid (SST, SST) \mid (SST)^*$
$ST ::= (SST)$
$\quad \mid (ST \cup ST) \mid (ST; ST) \mid (ST;)^*$
$STT ::= nill \mid tn \mid tb$
$\quad \mid (STT \cup STT) \mid (STT, STT) \mid (STT)^*$
$TT ::= (STT)$
$\quad \mid (TT \cup TT) \mid (TT; TT) \mid (TT;)^*$

Expresii

$V ::= x : ST \mid x : TT$
$\quad \mid V(k) \mid V.k \mid V.[k] \mid V@k \mid V@[k]$
$E ::= n \mid V \mid E + E \mid E * E \mid E - E \mid E/E$
$B ::= b \mid V \mid B\&\&B \mid B||B \mid !B \mid E < E$

Programe

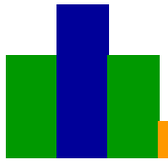
$W ::= nill \mid new\ x : SST \mid new\ x : STT$
$\quad \mid x := E \mid if(B)\{W\}else\{W\}$
$\quad \mid W; W \mid while(B)\{W\}$
$M ::= module\{listen\ x : STT\}\{read\ x : SST\}$
$\quad \{\ W\ \}\{speak\ x : STT\}\{write\ x : SST\}$
$P ::= nill \mid M \mid if(B)\{P\}else\{P\}$
$\quad \mid P; P \mid P\#P \mid P\$P$
$\quad \mid while\_t(B)\{P\} \mid while\_s(B)\{P\}$
$\quad \mid while\_st(B)\{P\}$



AGAPIA v0.1: Sintaxa

Comentarii:

- limbajul este intenţionat făcut *simplu*
  - nu are compunere şi while “generale”
- *scoping:*
  - exportă doar variabilele din `listen/read/speak/write`
- *tipuri* pentru *interfeţe spaţiale*:
  - se pleacă cu întregi şi booleeni sn, sb,
  - apoi, folosim $\cup$, ‘,’, $(\_)^*$ pentru *interfeţe de procese*
  - apoi, folosim $\cup$, ’;’, $(\_;)^*$ pentru *interfeţe de sisteme*
- *tipuri* pentru *interfeţe temporale* - similar
- notaţiile $V(k), V.k, V.[k], V@k, V@[k]$ se folosesc pentru a accesa *componentele unui tip*



AGAPIA v0.1: Sintaxa

..Comentarii:

- se construiesc natural *expresii*, *programe while* uzuale, *module*, şi *rv-programe structurate*
- avem un *format restricţionat*:
  - *întâi*, constuim programe while simple
  - *apoi*, module
  - *în final*, programe AGAPIA v0.1 (aplicând instrucţiunile de programare rv structurată)
- limbajul este *invariant* la dualitatea spaţiu-timp: se poate uşor defini un operator de dualitate care transformă un program AGAPIA v0.1 P întru-un program AGAPIA v0.1 $P^\vee$ astfel încât $P = P^{\vee\vee}$

Exemplu: Detectia terminarii

Exemplu: Un program pentru detecţia terminării într-un sistem distribuit

```
P= I1# for_s(tid=0;tid<tm;tid++){I2}#
   $ while_st(!(token.col==white && token.pos==0)){
      for_s(tid=0;tid<tm;tid++){R}}
```

where:

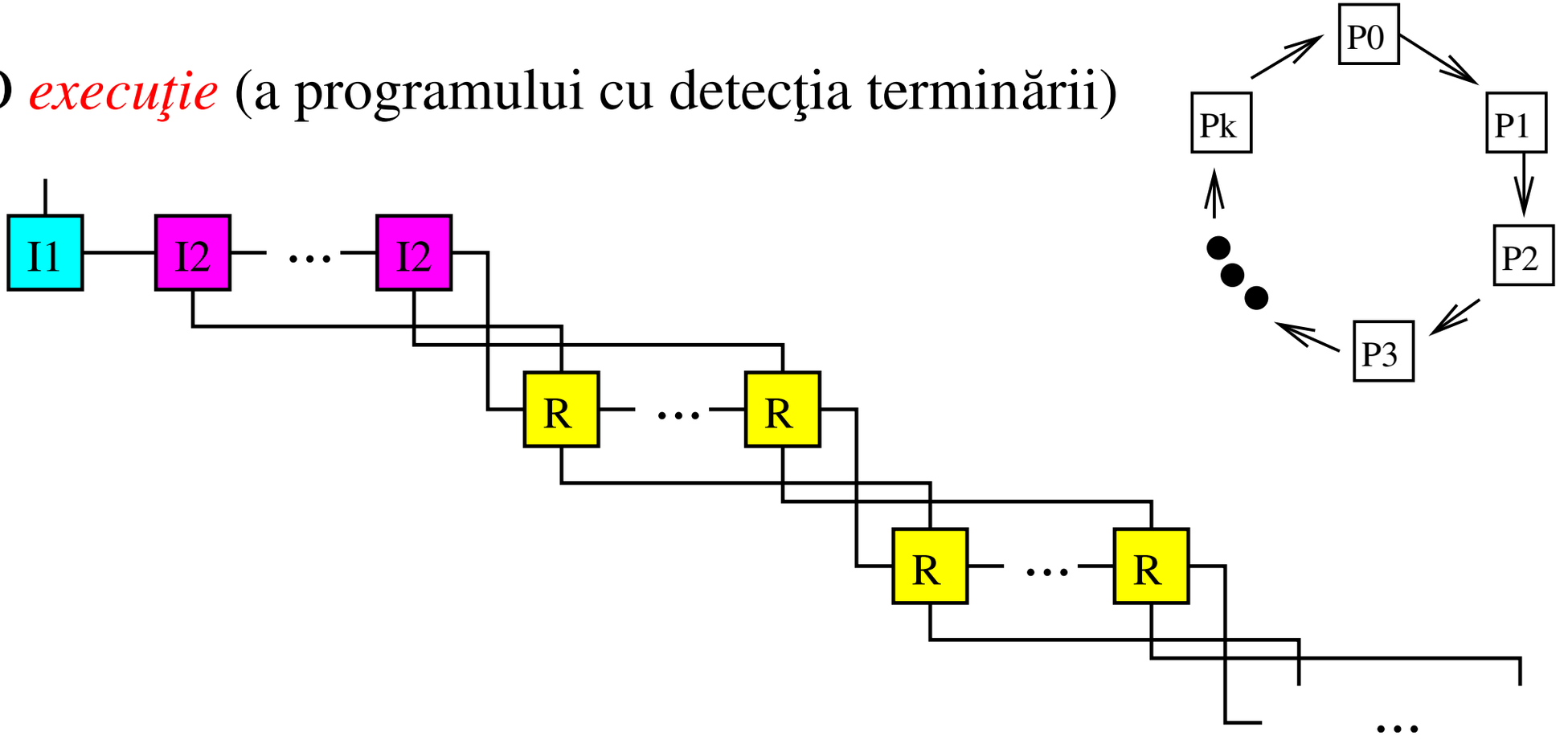
```
I1= module{listen nill}{read m}{
   tm=m; token.col=black; token.pos=0;
   }{speak tm,tid,msg[ ],token(col,pos)}{write nill}

I2= module{listen tm,tid,msg[ ],token(col,pos)}
   {read nill}{
   id=tid; c=white; active=true; msg[id]=null;
   }{speak tm,tid,msg[ ],token(col,pos)}
   {write id,c,active}
```

..Exemplu: Detectia terminarii

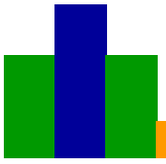
```
R=module{listen tm,tid,msg[ ],token(col,pos)}
   {read id,c,active}{
   if(msg[id]!=emptyset){ //take my jobs
      msg[id]=emptyset;
      active=true;}
   if(active){ //execute code, send jobs, update color
      delay(random_time);
      r=random(tm-1);
      for(i=0;i<r;i++){ k=random(tm-1);
        if(k!=id){msg[k]=msg[k]∪{id}};
        if(k<id){c=black};}
      active=random(true,false);}
   if(!active && token.pos==id){ //termination
      if(id==0)token.col=white;
      if(id!=0 && c==black){token.col=black;c=white};
      token.pos=token.pos+1[mod tm];}
   }{speak tm,tid,msg[ ],token(col,pos)}
   {write id,c,active}
```

..Exemplu: Detectia terminarii

O *execuţie* (a programului cu detecţia terminării)




```
I1# for_s(tid=0;tid<tm;tid++){I2}#
$ while_st(!(token.col==white && token.pos==0)){
    for_s(tid=0;tid<tm;tid++){R}}
```



AGAPIA v0.1: Sintaxa

Sintaxa lui AGAPIA v0.1:

Tipuri pentru interfeţe

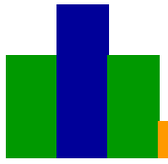
Folosim doi separatori “,” şi “;”

Pe interfeţe spaţiale:

- “,” separă tipurile utilizate *într-un proces*
- “;” separă tipurile utilizate în *procese diferite*

Pe interfeţe temporale:

- “,” separă tipurile utilizate folosite *într-o transacţie*
- “;” separă tipurile utilizate în *tranzacţii diferite*



Tipuri pentru interfete

Tipurile spaţiale simple sunt definite prin:

$$SST ::= nill \mid sn \mid sb \mid (SST \cup SST) \mid (SST, SST) \mid (SST)^*$$

(“,” - asociativ cu “nill” element neutru; “$\cup$” - asociativ)

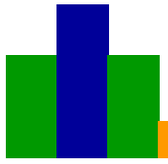
Exemplu:

$$((((sn)^*)^*, sb, (sn, sb, sn)^*)^*, (sb \cup sn))$$

reprezintă structura de date (pentru *un proces*)

```
x:  struc1[], where
    struc1 = ( a:  Int[][],
               b:  Bool,
               c:  struc2[], where
                   struc2 = (p:Int, q:Bool, r:Int)
             ),
y:  Bool or Int
```

Tipurile temporale simple — similar



Tipuri pentru interfete

Tipurile spaţiale sunt definite prin:

$$ST ::= nill \mid (SST) \mid (ST \cup ST) \mid (ST;ST) \mid (ST;)^*$$

(“;” - asociativ cu “nill” element neutru; “$\cup$” - asociativ)

Exemplu:

$$((sn)^*)^*;nill;sb;((sn)^*;)^*$$

reprezintă *o colecţie de procese* (A,B,C,D), unde

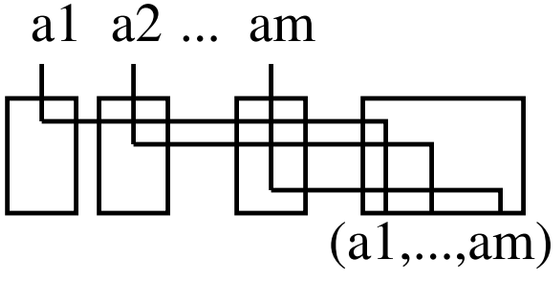
- A este *un proces* folosind un vector de întregi
- B este *un proces* fară date spaţiale de intrare
- C este *un proces* cu o variabilă booleană
- D este *un vector de procese*, fiecare proces utilizând un vector de întregi

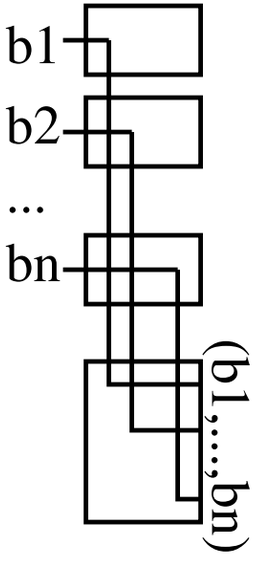
Tipurile temporale — similar

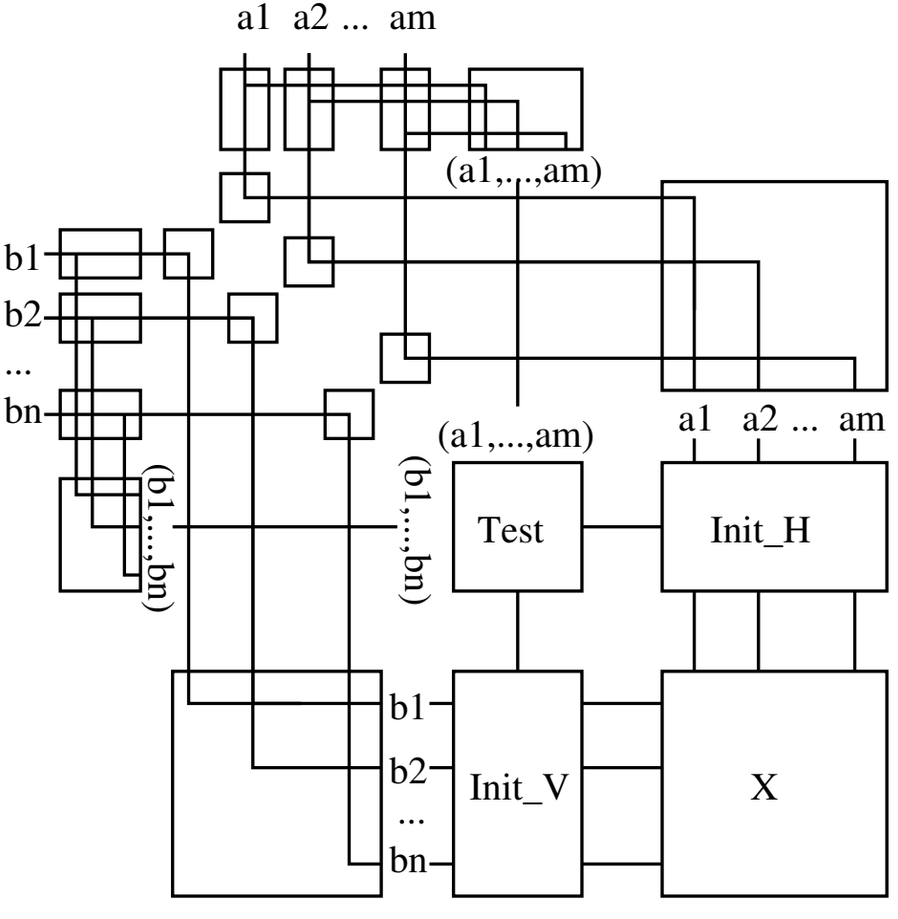
Interface types

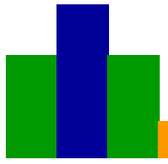
Restructurarea interfeţelor

- tipurile interfeţelor pot fi modificate cu morfisme speciale
- exemple $(sn;)^* \mapsto (sn)^*$ şi $(tn;)^* \mapsto (tn)^*$ (stânga)









AGAPIA v0.1: Sintaxa

Expresii

Variabile

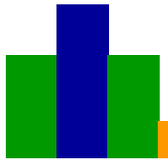
$$V ::= x : ST \mid x : TT \mid V(k) \mid V.k \mid V.[k] \mid V@k \mid V@[k]$$

Expresii arithmetice

$$E ::= n \mid V \mid E + E \mid E * E \mid E - E \mid E/E$$

Expresii booleene

$$B ::= b \mid V \mid B\&\&B \mid B||B \mid !B \mid E < E$$



..AGAPIA v0.1: Sintaxa

Programe

Programe while simple

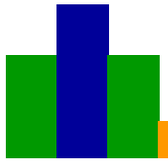
$$W ::= nill \mid new\ x : SST \mid new\ x : STT \mid x := E \mid if(B)\{W\}else\{W\} \mid W;W \mid while(B)\{W\}$$

Module

$$M ::= module\{listen\ x : STT\}\{read\ x : SST\}\{\ W\ \}\{speak\ x : STT\}\{write\ x : SST\}$$

Programe Agapia v0.1

$$P ::= nill \mid M \mid if(B)\{P\}else\{P\} \mid P;P \mid P\#P \mid P\$P \mid while\_t(B)\{P\} \mid while\_s(B)\{P\} \mid while\_st(B)\{P\}$$



..AGAPIA v0.1: Sintaxa

Compunere şi while temporale (ori verticale)

- notate ";" şi *while_t*
- compunere de module/programe via interfeţe spaţiale (compunere "uzuală")

Compunere şi while spaţiale (ori orizontale)

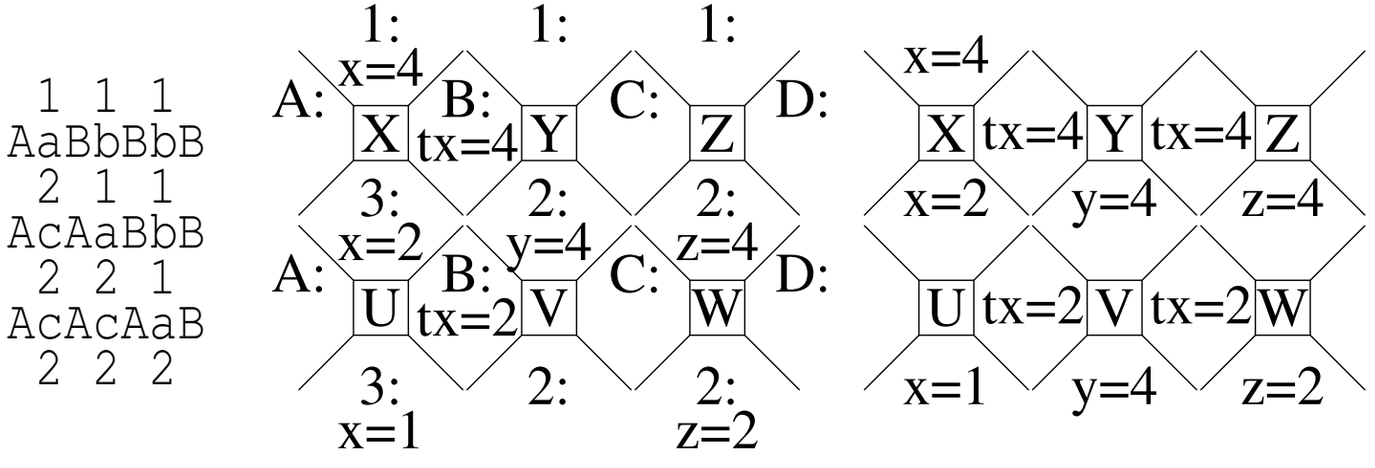
- notate "#" şi *while_s*
- compunere de module/programe via interfeţe temporale

Compunere şi while spatio-temporale (ori diagonale)

- notate "$" şi *while_st*
- compunere de module/programe folosind şi interfeţele spaţiale, şi pe cele temporale

Scenarii

Scenarii:



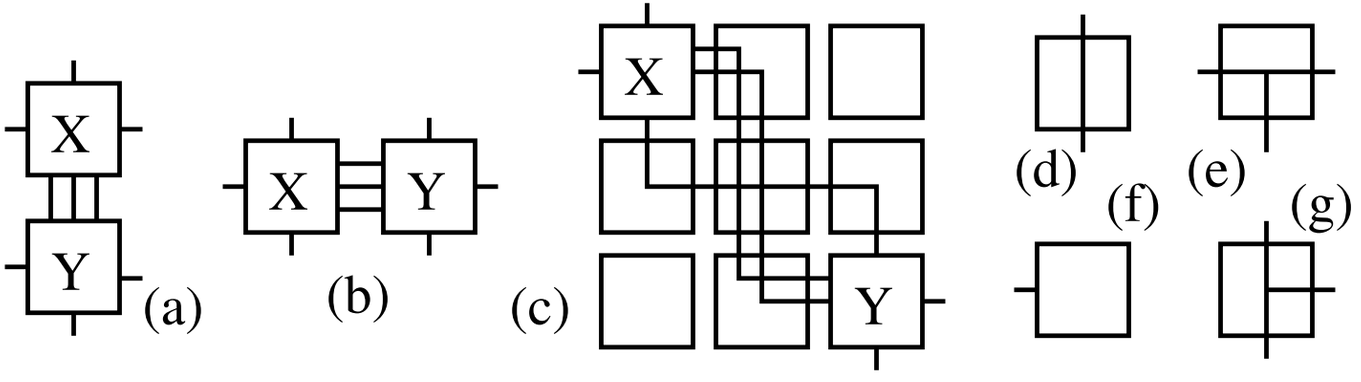
(1) (2) (3)

- în stânga este un scenariu abstract
- în mijloc avem un *rv-scenariu* (scenariu de rv-program):
  - sunt prezente etichete de stări şi clase
- în dreapta avem un *scenariu* (de rv-program structurat):
  - fără etichete de stări şi clase
  - cu date complet specificate în jurul fiecărei celule

Scenarii

Operaţii cu scenarii:

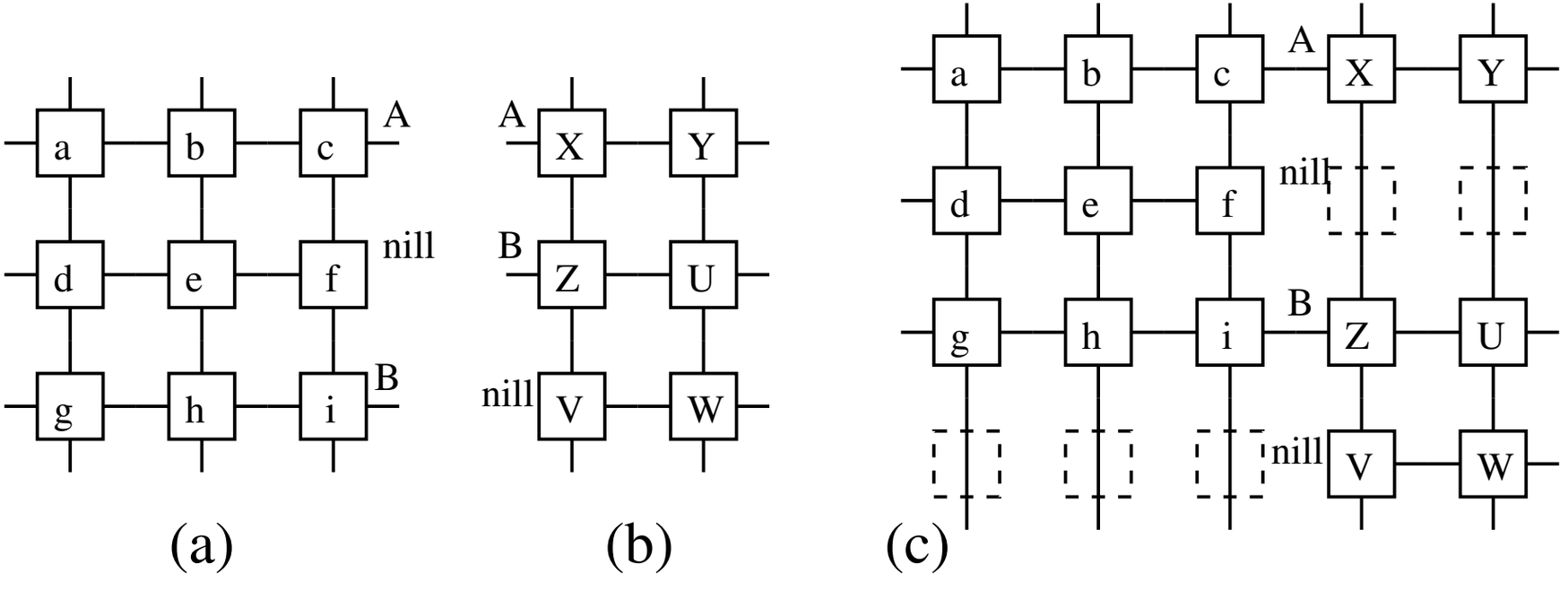
- Figură (vezi Lecţia 3 pentru comentarii, mai multe detalii, etc.)



..Operaţii cu scenarii

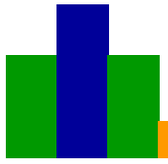
..Operaţii cu scenarii:

- Detalii la compunerea orizontală



(a) (b) (c)

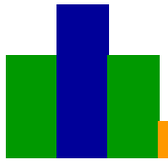
- Proceduri similare se folosesc şi la compunerile verticale şi diagonale



Exemplu: Detectia terminarii

Dual-pass ring termination detection protocol: Implementăm protocolul de terminare în ring descris în Lecţia 6

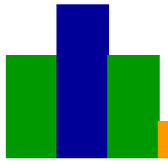
- P_0 devine alb când termină, generează un jeton alb, şi îl pasează lui P_1
- Jetonul trece de la un proces P_i la următorul în ring când P_i se termină, dar culoarea se poate schimba: dacă P_i a trimis un job unui proces P_j cu $j < i$, atunci a devenit negru; un proces negru pasează un jeton negru, altfel îl pasează cum l-a primit; după pasare, procesul P_i devine alb;
- Dacă P_0 primeşte un jeton negru, pasează un jeton alb (la terminare); dacă primeşte un jeton alb, toate procesele s-au terminat



..Exemplu: Detectia terminarii

Structurile de date:

- avem `m` procese `0,...,m-1`; folosim variabilele spaţiale `id : sInt`, `c : {white,black}`, `active : sBool` şi pe cele temporale `tm, tid : tInt`, `msg : tIntSet[]`
- variabilele spaţiale `id, c, active` reprezintă identitatea procesului, culoarea, şi starea activ/pasiv; variabilele temporale sunt: (i) `tm, tid` - versiuni temporale pentru `m, id`; (ii) `msg[ ]` - vector de mulţimi, unde `msg[k]` conţine `id`-ul proceselor destinaţie pentru mesajele încă nerecepţionate trimise de procesul `k`; (iii) `token.col` $\in$ `{white,black}` - culoarea jetonului; şi (iv) `token.pos` - id-ul procesului care are jetonul



..Exemplu: Detectia terminarii

Algorithmul:

- programul iniţializează reţeaua, construind `m` procese şi setându-le `id`, `c`, `active`; în plus, în această fază (tranzacţie) $\texttt{msg[i]} = \emptyset$, pentru $0 \leq \texttt{i} < \texttt{m}$, iar culoarea/poziţia jetonului sunt black/0.

- după iniţializare şi până se termină, fiecare proces execută codul `R`: Fie că este activ ori nu, verifică dacă a primit job-uri; dacă da, le colectează şi devine/rămne activ. Când este activ, execută cod, trimite noi mesaje, şi aleator trece în starea activ/pasiv. Dacă are jetonul şi s-a terminat (i.e., este pasiv) îl trimite cu culoarea corespunzătoare procesului următor

..Exemplu: Detectia terminarii

Codul:

```
P= I1# for_s(tid=0;tid<tm;tid++){I2}#
   $ while_st(!(token.col==white && token.pos==0)){
      for_s(tid=0;tid<tm;tid++){R}}
```

where:

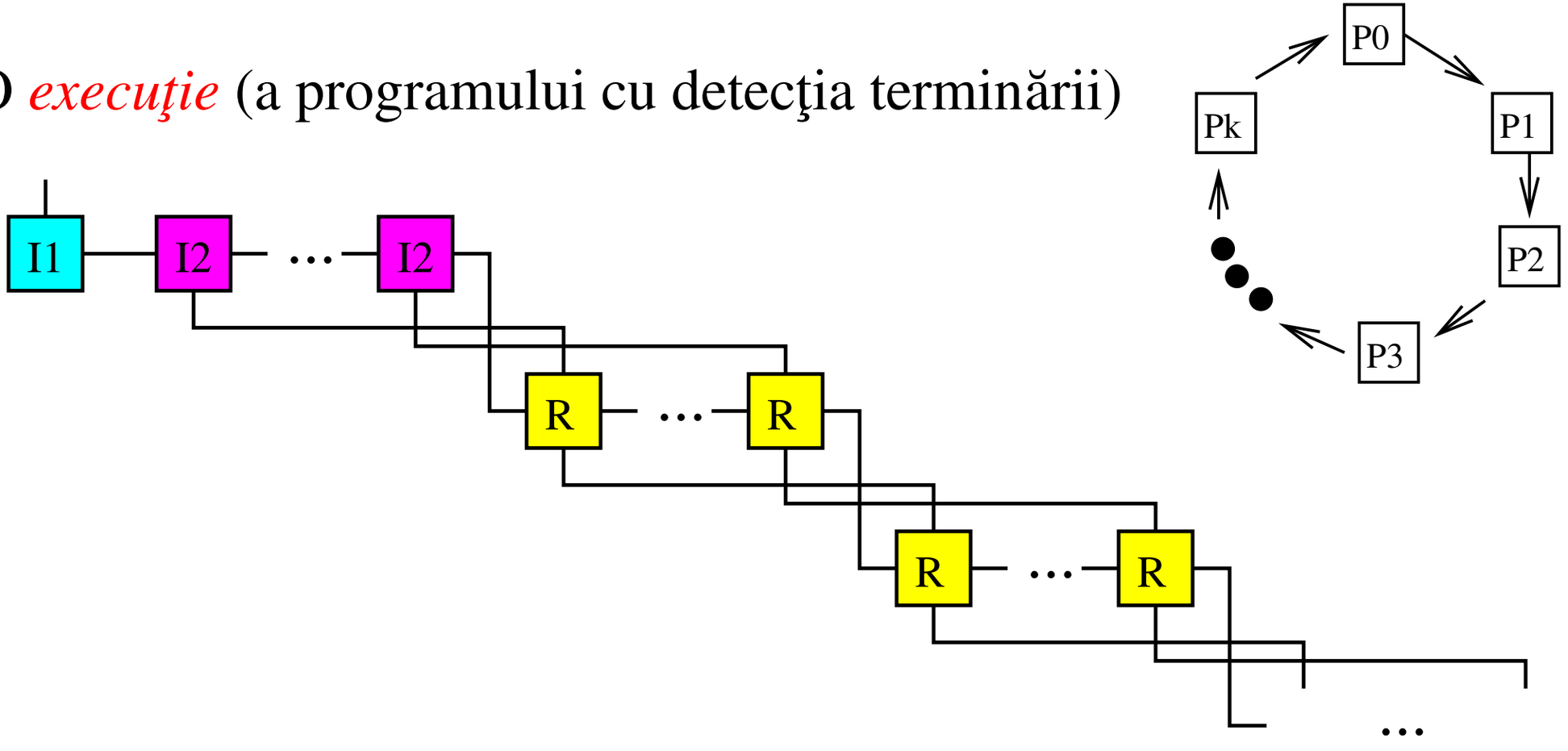
```
I1= module{listen nill}{read m}{
    tm=m; token.col=black; token.pos=0;
    }{speak tm,tid,msg[ ],token(col,pos)}{write nill}

I2= module{listen tm,tid,msg[ ],token(col,pos)}
    {read nill}{
    id=tid; c=white; active=true; msg[id]=null;
    }{speak tm,tid,msg[ ],token(col,pos)}
    {write id,c,active}
```

..Exemplu: Detectia terminarii

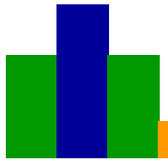
```
R=module{listen tm,tid,msg[ ],token(col,pos)}
   {read id,c,active}{
   if(msg[id]!=emptyset){ //take my jobs
      msg[id]=emptyset;
      active=true;}
   if(active){ //execute code, send jobs, update color
      delay(random_time);
      r=random(tm-1);
      for(i=0;i<r;i++){ k=random(tm-1);
        if(k!=id){msg[k]=msg[k]∪{id}};
        if(k<id){c=black};}
      active=random(true,false);}
   if(!active && token.pos==id){ //termination
      if(id==0)token.col=white;
      if(id!=0 && c==black){token.col=black;c=white};
      token.pos=token.pos+1[mod tm];}
   }{speak tm,tid,msg[ ],token(col,pos)}
   {write id,c,active}
```

..Exemplu: Detectia terminarii

O *execuţie* (a programului cu detecţia terminării)



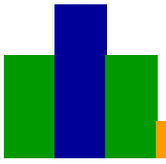

```
I1# for_s(tid=0;tid<tm;tid++){I2}#
$ while_st(!(token.col==white && token.pos==0)){
    for_s(tid=0;tid<tm;tid++){R}}
```



Compilarea rv-programelor

Compilarea rv-programelor (situaţie curentă)

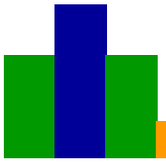
- translatare: rv-programe structurate $\mapsto$ rv-programe
  - ok (teoretic)
- simulator pentru execuţia rv-programelor
  - ok (teoretic şi practic)
- optimizări
  - nu încă
- studii de: limbaje de asamblare, extensii de arhitectura
  - nu încă



Calcul interactiv

Cuprins:

- Sisteme interactive: Generalitati
- Specificatii spatio-temporale
- RV-programe (nestructurate)
- RV-programe structurate; AGAPIA v0.1
- *Concluzii, diverse, etc.*



Concluzii, diverse, etc.

a se insera...