Concepte fundamentale ale limbajelor de programare

Definirea limbajelor de programare

Curs 03

conf. dr. ing. Ciprian-Bogdan Chirila

Universitatea Politehnica Timisoara
Departamentul de Calculatoare si Tehnologia Informatiei

March 6, 2023

Cuprins

Limbajul de programare

- Este o notație formală
- Forma și sensul sunt descrise de un set de reguli
- Regulile stabilesc:
  - Când un program este scris corect
  - Ce se va întampla la execuție
- Regulile sintatice definesc sintaxa limbajului de programare
- Regulile semantice definesc semantica limbajului de programare

Definiția unui limbaj de programare

- $L =< S_m, S_t, f : S_t \rightarrow S_m >$
- S_m este semantica limbajului
- S_t este sintaxa limbajului
- f este funcția de asociere dintre sintaxă și anumita semantică

Cuprins

Metode de definire formala ale limbajelor de programare

- Se definește un alfabet A format din simbolurile de bază
- Se definește mulțimea A^* conținând toate șirurile de simboluri posibile ce pot fi construite din elementele mulțimii A
- Se definește un set de reguli pentru a selecta mulțimea de programe corecte $P \subseteq A^*$
- Specificația semantică a fiecărui element $p \in P$

Sintaxa

- Regulile de sintaxa generează o mulțime infinită de propoziții
- Doar un subset dintre ele sunt corecte din punct de vedere semantic
- Propozițiile sunt formate din simboluri
- Simbolurile sunt formate din caractere ce respectă regulile lexicale
- Regulile lexicale aparțin sintaxei limbajului
- Toate simbolurile formează vocabularul limbajului de programare
  - Identificatori, cuvinte cheie
    - *begin*, *end* în Pascal
    - $+$, $++$, $<=$, in C
  - Constante intregi, constante float, constante string

Gramatici

- Toate regulile sintactice ale unui limbaj formează gramatica
- Cum se scrie o gramatică?
- BNF Bachus Naur Form
  - Folosită pentru Algol 60
- BNF extins sau EBNF
  - metalimbaj
  - inseamna un limbaj utilizat spre a defini un alt limbaj

Metasimboluri EBNF

- : := înseamnă definit ca
- | înseamnă sau
- < si > sunt utilizate pentru neterminale
- [si] sunt utilizate pentru secvențe opționale
- { si } sunt utilizate pentru secvențe ce se repetă de zero sau mai multe ori

Sintaxa

- sintaxa este un set de relații sau reguli EBNF
- o relație definește
  - un neterminal specificat în stanga semnului ::=
  - neterminale sau terminale în partea dreaptă trebuie definit într-o relație diferită
- terminale sau simboluri de limbaj
- fiecare neterminal utilizat în partea dreaptă trebuie definit într-o relație diferită
- o gramatica completă trebuie sa defineasca toate neterminalele
- un neterminal se defineste ca simbol de start al gramaticii
- de obicei el se numește `<program>`

Programul

- este un șir de simboluri
  - simbolurile se mai numesc terminale sau atomi
- este corect din punct de vedere sintactic
  - dacă șirul de simboluri poate fi derivat pe baza regulilor gramaticale începând cu simbolul de start
  - dacă sunt consumate astfel toate simbolurile

Exemplu de gramatică

```
<expression> ::= <term> { +|- <term> }
<term> ::= <factor> { *|: <factor> }
<factor> ::= number | identifier | ( <expression> )
<assignment> ::= identifier := <expression>
<instructions> ::= <assignment> { ; <assignment> }
<program> ::= prog identifier; <instructions> end.
```

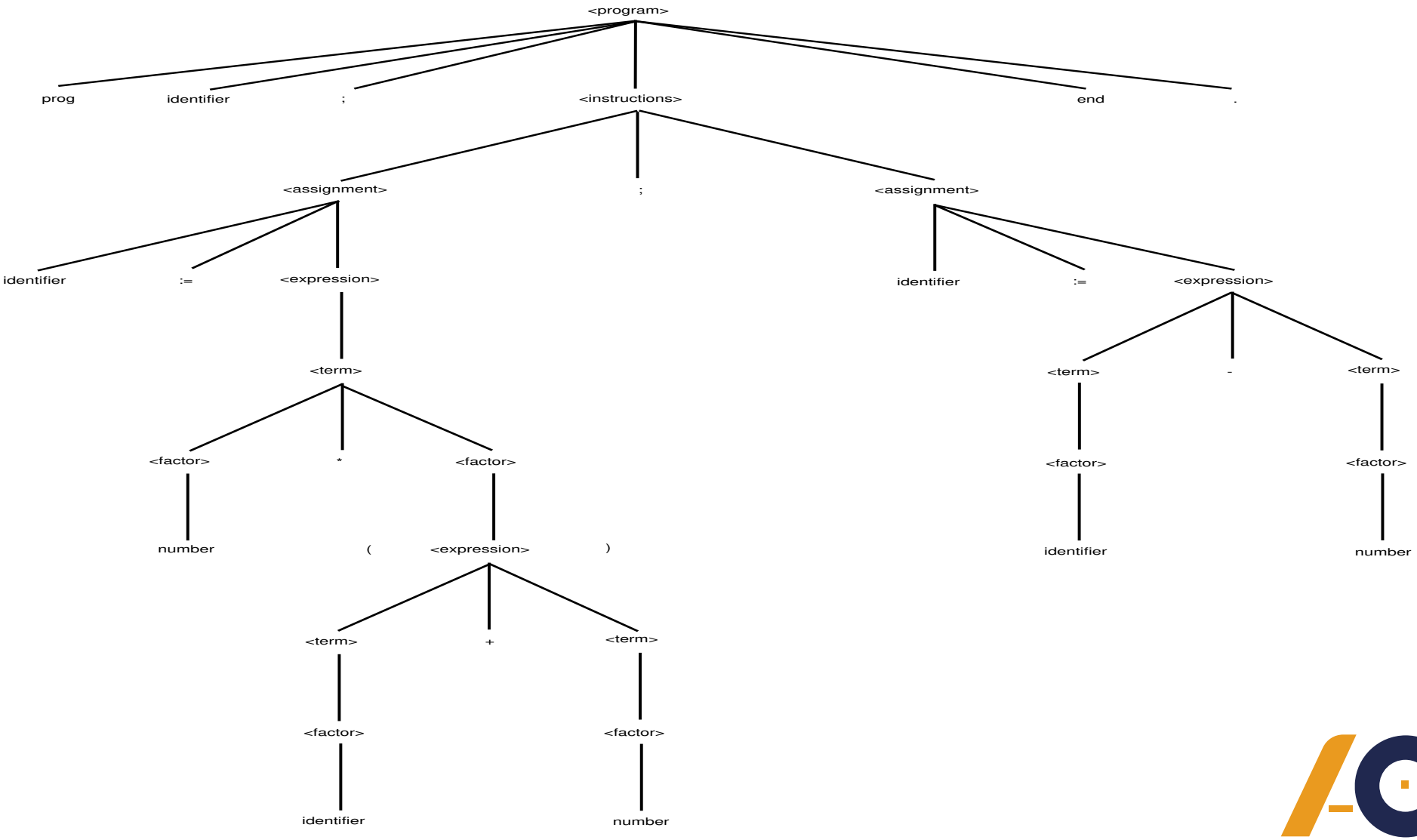
Exemplu de program

```
prog example;
 a:=2*(x+3);
 b:=a-1
end.
```

Exemplu de program

- Este corect din punct de vedere sintactic dacă poate fi derivat pe baza regulilor pornind de la neterminalul program
- Procesul de derivare poate fi ilustrat prin desenarea unui arbore în care:
  - Radacina este simbolul de start
  - Nodurile interne sunt neterminale
  - Nodurile frunză sunt terminalele
- Astfel rezultă arborele de sintaxa

Procesul de derivare



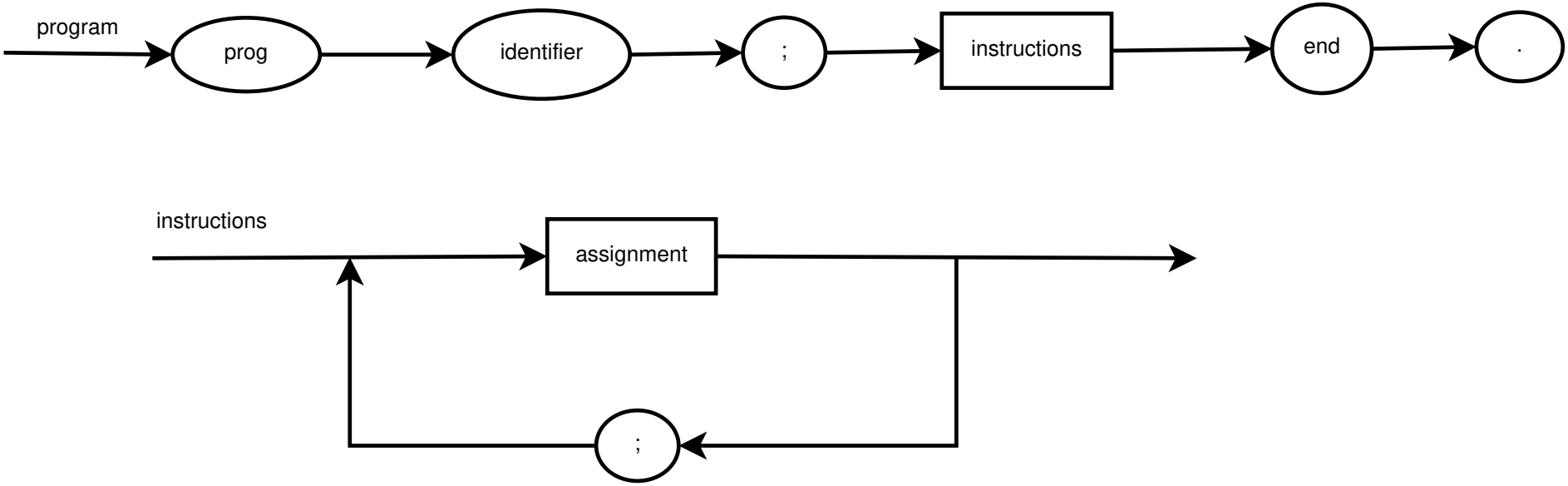
Analiza sintactică

- verifică corectitudinea din punct de vedere sintactic a unui program
- opereaza de jos în sus
  - pornește de la simboluri
  - se identifică și se înlocuiesc secvențe egale cu părțile drepte ale regulilor cu neterminalele lor
  - se repetă procesul până când se ajunge la simbolul de start al gramaticii
- opereaza de sus în jos
  - se pornește de la simbolul de start al gramaticii
  - se înlocuiesc neterminalele cu conținutul regulilor gramaticale
  - se repetă procesul până când nu mai există niciun neterminal de înlocuit

Diagrama de sintaxă

- O succesiune de simboluri este corectă dacă poate fi generată prin traversarea diagramei de la început până la sfârșit
- când se întâlnește un dreptunghi
  - atunci trebuie verificat neterminalul corespunzător
- cand se intâlneste un cerc sau o elipsa
  - atunci trebuie verificat terminalul corespunzător

Exemplu de diagrama de sintaxă



Expresii regulate

- Regulile lexicale pot fi exprimate prin intermediul expresiilor regulate
- Fiecare expresie regulată e generează o mulțime de șiruri S
  - formate din literele unui alfabet A
  - aplicând un set de operatori

Expresii regulate

- Să presupunem ca S, $S1$, $S2$ sunt mulțimi de șiruri de caractere
- Reuniune: $S_1 \cup S_2 = \{s | s \in S_1 \text{ or } s \in S_2\}$
- Produs sau concatenare: $S_1 S_2 = \{s_1 s_2 | s_1 \in S_1 \text{ and } s_2 \in S2\}$
- Ridicarea la putere:

$$S^n = \begin{cases} \{\epsilon\}, & \text{daca n=0, } (\epsilon \text{ este sirul vid}) \\ S^{n-1} S \forall n \in N, & \text{daca } n \geq 1 \end{cases}$$

Expresii regulate

- Închidere Kleene sau stea:

$$S^* = \bigcup_{i=0}^{\infty} S^i$$

- Închidere pozitivă sau plus:

$$S^+ = \bigcup_{i=1}^{\infty} S^i$$

- de exemplu:
  - $L = \{A, B, C, \ldots Z, a, b, \ldots, z\}$
  - $C = \{0, 1, \ldots, 9\}$

Definirea de noi mulțimi

- Mulțimea tuturor literelor și a cifrelor $L \cup D$
- Mulțimea tuturor șirurilor de caractere LD unde:
  - Primul caracter este o literă
  - Al doilea caracter este o cifră
- Mulțimea șirurilor de 4 litere L^4
- Mulțimea șirurilor de litere de orice lungime incluzând șirul de caractere vid L^*
- Mulțimea șirurilor de cifre conținând cel puțin o cifră D^+

Construcția unei expresii regulate

- Pornim de la un alfabet A
- ϵ este o expresie regulată
- a este o expresie regulată
- e1, e2, e sunt expresii regulate ce generează mulțumile $S1$, $S2$, S
- pe aceste mulțimi putem aplica diferiți operatori
- rezultatul va fi o expresie regulată

Operatori pentru expresii regulate

- reuniune $(e1)|(e2)$ rezultă mulțimea $S1 \cup S2$
- produs sau concatenare $(e1)(e2)$ rezultă mulțimea $S1S2$
- stea $(e)^*$ rezultă mulțimea $(S)^*$
- toți operatorii sunt asociativi la stanga
- prioritatea operatorilor
  - de la cea mai mare la cea mai mică
  - stea, produs, reuniune

Operatori pentru expresii regulate

- unul sau mai mulți
  - operatorul "plus" $+$
  - expresia ee^* este echivalentă cu e^+
- zero sau unul
  - operatorul "semnul intrebarii" ?
- clase de caractere
  - notația $[c1c2c3c4]$ exprimă expresia regulată $c1|c2|c3|c4$
  - $a|b|\ldots|z$ va deveni $[a-z]$

Exemple

- `litera(litera | cifra)*`
  - expresia regulată pentru identificatori
  - `L(L ∪ C)*` din exemplele anterioare
- `cifra -> [0-9]`
- `litera -> [A-Z,a-z]`
- `identificator -> litera(litera|cifra)*`
- `cifre -> cifra+`
- `exponent -> ((E|e)(+|-)?cifre)?`
- `parte_fractionara -> (.cifre)?`
- `numar -> cifre parte_fractionara exponent`
- când numele de reguli sunt folosite în partea dreaptă avem de a face cu o definiție regulată

Cuprins

Semantica

- Reguli semantice
  - Înțelesul asociat cu construcțiile sintactice corecte
- Descrierea sintaxei
  - BNF
  - EBNF
- Descrierea semanticii
  - Coexistă o serie de metodologii
  - Una perfect satisfăcătoare este încă subiect de cercetare

Semantica unui limbaj de programare

- Este descrisă în limbaj natural prin:
  - texte, desene, diagrame
- Sunt mai mult sau mai puțin riguroase
- Sunt bune pentru învățare
  - an I PC - semantica limbajului C: variabile, tipuri, instrucțiuni
  - an I TP - semantica conceptelor de programare C: liste, fisiere etc.
  - an II POO - semantica conceptelor de programare OOP in Java: clase, obiecte, mostenire etc.
- Ambiguitățile sunt clarificate prin experimente

De ce e nevoie de descrierea formală a semanticii?

- Din cauză că limbajele de programare
  - au o raspandire largă
  - tind să devină complexe și diverse
- Din cauză că aplicațiile
  - sunt complexe, mari și diverse
  - cer fiabilitate sporită
- soluția: notație formală, matematică
  - fără ambiguități
  - mai dificil de accesat (înțeles)
  - necesită pregătire specială pentru a descifra formalismele

Avantajele formalismelor

- Se evită lacunele în definirea limbajelor
  - Lacunele sunt foarte probabile în definiția informală
- Reprezintă documentație de referință pentru programator
  - Programatorul își poate clarifica diferite probleme când citește definiția informală
- Documentație de referință pentru implementare
  - Pentru echipa de implementare a limbajului de programare
  - Poate fi utilizată în validarea și omologarea (standardizarea) implementării

Avantajele formalismelor

- Reprezintă baza formală pentru verificarea automată a programelor
  - Algoritmii de verificare formală necesită o definiție riguroasa a limbajului de programare
- Independența la implementare
  - Formalismele garantează independența limbajului față de implementare

Criterii de apreciere pentru metodele formale

- Completitudinea
  - Capabilitatea metodei de a acoperi toate problemele de sintaxă si semantică
- Simplicitatea
  - Ușurința cu care se poate crea un model indiferent cât de complex este limbajul

Criterii de apreciere pentru metodele formale

- Claritatea
  - Înțelegerea facilă a definițiilor
  - Descrierea naturală a limbajului de programare
- Expresivitatea la erori
  - Capabilitatea metodei de a detecta erorile de program

Criterii de apreciere pentru metodele formale

- Flexibilitatea
  - Capabilitatea metodei de a defini locații unde restricțiile sau opțiunile sunt lăsate libere pentru implementatori
- Modificabilitatea
  - Capabilitatea metodei de a permite în mod facil modificări în descrierea anterioară a unui limbaj de programare
  - Este importantă în faza de definiție a limbajului de programare

Metode formale pentru semantica limbajelor de programare

- 2 metode
  - intuitive
  - bazate pe concepte de traducere a programelor
- 2 metode
  - matematice
  - cu o bază teoretică puternică
- Compararea metodelor
  - utilizand criteriile prezentate anterior

Semantica operațională

- Este definita de efectele pe care construcțiile de limbaj le au asupra unui procesor real sau virtual
- Semantica unei instrucțiuni este definită prin:
  - Cunoașterea stării calculatorului
  - Executarea unei instrucțiuni
  - Examinarea stării noi a calculatorului

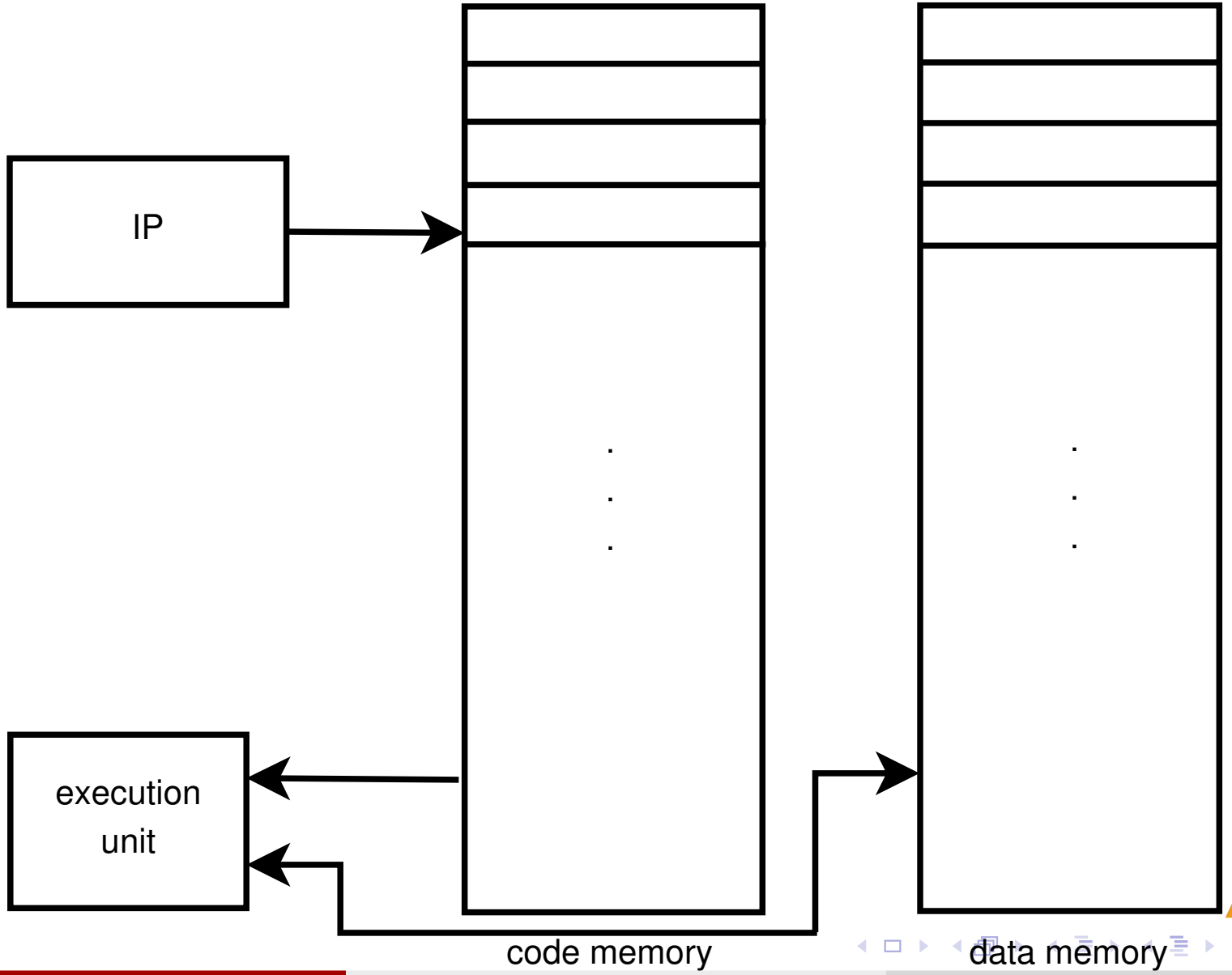
Semantica operațională

- deoarece arhitecturile calculatoarelor reale sunt foarte complexe
- sunt utilizate mașinile virtuale în schimb
- interpretoarele software execută instrucțiunile virtuale
- mașinile virtuale pot fi create astfel încât semantica limbajelor de programare să poată fi exprimată ușor prin instrucțiuni virtuale
- setul de instrucțiuni virtuale trebuie să fie suficient de simplu pentru a putea fi implementat pe orice mașină hardware

Aplicarea metodei semanticii operaționale

- Definirea si implementarea unei mașini virtuale MV
- Crearea unui translator care să convertească instrucțiunile unui limbaj L în instrucțiunile mașinii virtuale MV
- Schimbările de stare produse în mașina virtuală prin executarea codului virtual rezultat prin translatarea instrucțiunilor limbajului L definește semantica instrucțiunii

Structura unei mașini virtuale (MV)



Rularea mașinii virtuale

- Procesor virtual
  - pointer la instrucțiunea curentă PI
- Memorie pentru cod C
- Memorie pentru date D
- Ciclul mașinii virtuale
  - Se execută instrucțiunea referită de PI
  - Daca instrucțiunea curentă nu schimbă PI
    - Atunci PI este incrementat
    - Astfel este referită urmatoarea instrucțiune din memoria de cod C

Exemplu de program

```
for i:=first to last do
begin
...
end;
```

```
i=first;
loop: if i>last goto out
...
i:=i+1
...
goto loop
out: ...
```

Descrierea semanticii operaționale

- A fost utilizată pentru prima dată la IBM filiala Viena
- A fost utilizată pentru a defini semantica limbajului PL/I în anul 1969
- VDL - Vienna Definition Language
- Este bun pentru programatori și implementatori
- Nu este bazat pe un formalism matematic complicat
- Este bazat pe translatarea de algoritmi
- Semantica limbajului de programare este definită în termenii unui alt limbaj cunoscut de nivel scazut

Gramatici atributate

- Utilizate când procesul de translatare este dirijat de o gramatică
- Semantica poate fi specificată prin atașarea de atribute semantice simbolurilor gramaticale
  - Terminale
  - Neterminale
- Metoda a fost propusă de Donald Knuth în anul 1968

Gramatici atributate

- valorile atributelor sunt calculate pe baza expresiilor sau a funcțiilor ce sunt numite reguli semantice
  - ele sunt asociate regulilor gramaticale
- evaluarea regulilor semantice înseamnă analiza semantică
- acest proces este numit si translatare dirijată de sintaxă
- sunt mai multe asocieri posibile între regulile semantice si regulile gramaticale
  - definiții dirijate de sintaxa DDS

Definiții dirijate de sintaxă (DDS)

- Este o generalizare a gramaticii
- La fiecare simbol atasam un set de atribute
- Rezulta o gramatică atributată
- Reprezentarea atributelor
  - Numere
  - Șiruri de caractere
  - Tipuri
  - Locatii de memorie (pointeri)
- Atributele sunt calculate în timpul dezvoltării arborelui de sintaxă
- Valoarea atributului este calculată utilizând o regulă semantică asociată cu o producție aferentă

DDS pentru un calculator de birou

```
<line>::=<expression>nl
<expression>::=<expression>+<term>|<term>
<term>::=<term>*<factor>|<factor>
<factor>::=(<expression>)|number
```

Definiții dirijate de sintaxă

- definiția asociază la fiecare neterminal $< expression >$, $< term >$, $< factor >$ câte un atribut întreg cu numele val
- Pentru fiecare producție calculăm:
  - atributul val asociat cu neterminalul din partea stangă
  - bazându-ne pe valorile atributelor val ale neterminalelor din partea dreaptă

DDS pentru un calculator de birou

Grammar production	Semantic rules
<line>::=<expression>nl	print(<expression>.val)
<expression>::=<expression1>+<term>	<expression.val>:=<expression1>.val+<term>.val
<expression>::=<term>	<expression>.val:=<term>.val
<term>::=<term1>*<factor>	<term>.val:=<term1>.val*<factor>.val
<term>::=<factor>	<term>.val:=<factor>.val
<factor>::=(<expression>)	<factor>.val:=<expression>.val
<factor>::=number	<factor>.val:=number.lexval

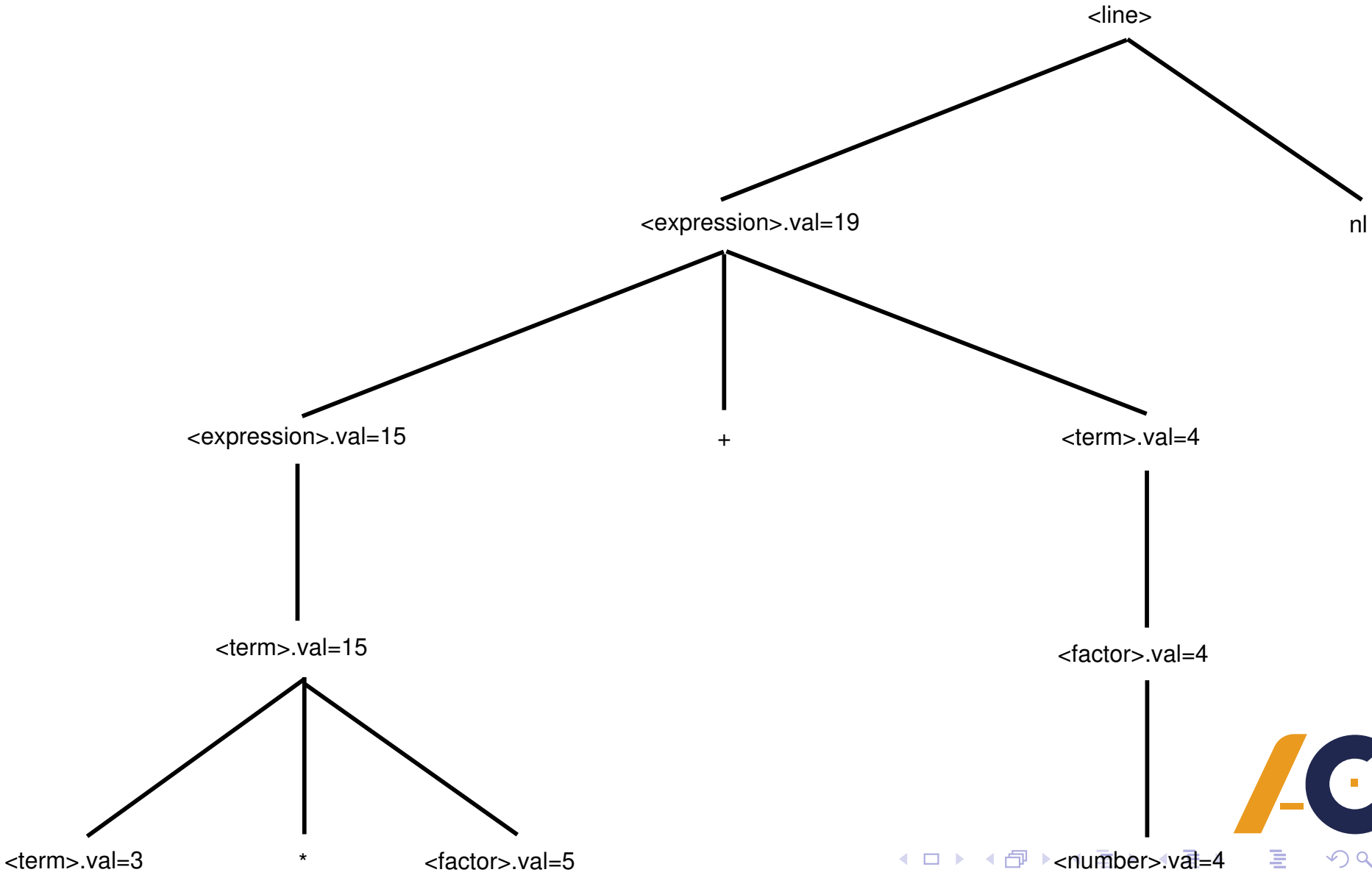
DDS pentru un calculator de birou

- atomul number are un atribut numit lexval
- regula de start afisează valoarea neterminalului $< \textit{expression} >$

Arbore de sintaxă adnotat

- Este un arbore de sintaxă care afisează atributele nodurilor
- Procesul este numit adnotarea arborelui de sintaxă
- O definiție ce utilizează doar atribute sintetizate este numită definiție S-atributată

Exemplu de arbore de sintaxă adnotat



Semantica axiomatică

- a traduce o instrucțiune corectă într-un meta-limbaj matematic real sau virtual
- o notație ce are reguli matematice bine definite
- trebuie determinat un set de reguli de translație între
  - Domeniul construcțiilor de limbaj
  - Forumelele matematice – meta-limbaj

Semantica axiomatică a meta-limbajului

- C.A.R. Hoare 1969
- își are rădăcinile in logica matematică
- bazată pe calculul de predicate
- predicatele
  - Sunt expresii logice aplicate variabilelor de program
  - Sunt utilizate pentru a exprima stările din procesul de calcul

Preconditii si postconditii

- Instrucțiunea S
- Predicatul P
  - Trebuie sa fie adevărat după executarea lui S
  - Este numit postcondiție pentru S
- daca predicatul Q
  - este adevărat
  - cand S se execută normal
  - și postcondiția P este adevărată
- atunci Q este numit precondiție pentru S și P

Exemplu

```
Notation:
Q {S} P
Example:
S: x:=y+1 (integers)
P: x>0 y=3 {x:=y+1} x>0
Q: y=3
------------
Q: y>-1  y>-1 {x:=y+1} x>0
```

Exemplu

- pentru
  - intrucțiunea S
  - predicatul P
- există multiple (un infinit) de precondiții disponibile
- una dintre ele este numită cea mai slabă precondiție
- Toate precondițiile Q o implică pe cea mai slabă precondiție W
- Pentru orice precondiție Q adevarată și precondiția W va fi adevarată

Semantica axiomatică

- $\forall$ preconditie $Q \rightarrow W$ (relația de implicare)
- $p \rightarrow q$ (inseamnă că oricând p este adevărat de asemenea și q este adevărat)
- $y = 3 -> y > -1$ este ADEVĂRAT
- $y > 0 -> y > -1$ este ADEVĂRAT
- $y > -5 -> y > -1$ este FALS
- doar cea mai slaba preconditie este importantă
- Pentru a exprima efectul construcției prin transformări de predicate
- se definește funcția axsem
  - $axsem(S, P) = W$
  - S – constructie de limbaj
  - P – postcondiție
  - W – cea mai slabă precondiție
- A defini semantica unui limbaj înseamna a defini funcția axsem pentru toate construcțiile sale

Funcția axsem pentru instrucțiunea de atribuire

- `axsem(x:=E,P)`=$P_{x \to E}$
- $P_{x \to E}$ este predicatul P unde toate aparițiile lui x au fost înlocuite cu E
- Pentru ca predicatul P să fie adevărat după ce x a primit valoarea E, înainte de atribuire predicatul obținut prin înlocuirea lui x cu E trebuie să fie adevărat
- $P_{x \to E}$ `{x:=E} P`
- `y>-1 {x:=y+1} x>0`
  - în `x>0` înlocuim pe `x` cu `y+1`
  - în `y+1>0` sau `y>-1`
  - semantica atribuirii este că dacă `y>-1` atunci `x>0`

Exemplu

- vom folosi funcția `axsem` pentru a găsi în ce condiții atribuirea `x:=x+3` va produce un rezultat `x>8`
- `axsem(x:=x+3,x>8)=x>5`
  - în `x>8` inlocuim pe `x` cu `x+3`
  - dacă `x>5` atunci după atribuire `x>8`
  - semantica atribuirii este că dacă `x>5` atunci `x>8`

Funcția axsem pentru o secvență de instrucțiuni

- dacă considerăm:
  - `axsem(S1,P)=Q`
  - `axsem(S2,Q)=R`
- atunci pentru secvența `S2;S1`
  - `axsem(S2;S1,P)=R`
- postcondiția creată de `S2` devine precondiție pentru `S1`
- `R S2 Q`
- `Q S1 P`
- după secvențiere avem `R S2 Q S1 P` sau `R S2;S1 P`

Funcția axsem pentru instrucțiunea if

- `if B then L1 else L2 endif`
- B conditie
- L1,L2 sunt secvente de instructiuni
- `axsem(instr-if,P)` $\Rightarrow$
  `B => axsem(L1,P)`
  și
  `not B => axsem(L2,P)`

Exemplu

- `if x>=y then max:=x else max:=y endif`
- dacă secvența de cod calculează valoarea max în mod corect
- atunci `(x>=y and max=x) or (y>x and max=y)` trebuie să fie adevarată
- daca P este postcondiția, atunci care este precondiția ?
- `(x>=y)` $\Rightarrow$ `((x>=y and x=x) or (y>x and x=y))` `and` `not(x>=y)` $\Rightarrow$ `((x>=y and y=x) or (y>x and y=y))` `= true`

Semantica denotațională

- dezvoltată de Christopher Strachey și Dana Scott în 1970
- S=<mem,i,o>
  - mem este o funcție ce reprezintă memoria
  - $Mem : Id \rightarrow Z \cup \{undef\}$
    - Id este mulțimea tuturor identificatorilor
    - Z este mulțimea tututor numerelor întregi
    - $undef$ este valoarea unui identificator nedefinit
  - i,o secvențe de intrare și de ieșire
    - valorile lor pot fi secvențe de întregi sau secvențe vide

Semantica denotațională

- Folosind această reprezentare fiecare construcție de limbaj este exprimată ca o funcție
- Funcțiile arată modificările produse de construcțiile de limbaj asupra stării sistemului
- Toate funcțiile și toate regulile de compoziție reprezintă definiția semantică a limbajului
- Meta-limbajul matematic pentru semantica denotațională este calculul funcțional

Expresia aritmetică

- $dsemEx : EX \times S \to Z \cup \{error\}$
  - S este mulțimea de stări
  - EX este mulțimea de expresii
- $dsemEx(E, s) = error$
  - dacă $s =< mem, i, o >$ si $mem(v) = undef$ pentru o variabilă v din expresia E; altfel
- $dsemEx(E, s) = e$
  - dacă $s =< mem, i, o >$ si e este rezultatul evaluării expresiei E după înlocuirea fiecărui identificator v din expresia E cu $mem(v)$
- se presupune că expresia
  - nu are efecte colaterale
  - nu au loc supradepășiri
  - nu sunt erori de tip

Instrucțiunea de atribuire

- $dsemAs : AS \times S \to S \cup \{error\}$
  - AS este mulțimea instrucțiunilor de atribuire
  - $dsemAs(x := E, s) = error$
    - dacă $dsemEx(E, s) = error$; altfel
  - $dsemAs(x := E, s) = s'$
    - unde $s =< mem, i, o >$ si $s' =< mem', i', o' >$
    - $i' = i$ $o' = o$
    - $mem'(y) = mem(y)$ pentru orice $y \neq x$
    - $mem'(x) = dsemEx(E, s)$

Instrucțiunea de citire

- $x \Leftarrow read$
- $dsemRd : RD \times S \cup \{error\}$
  - RD este mulțimea instrucțiunilor de citire
  - $dsemRd(x \leftarrow read, s) = error$
    - dacă $s =< mem, i, o >$ și i este void, altfel
  - $dsemRd(x \Leftarrow read, s) = s'$
    - unde $s =< mem, i, o >$ si $s' =< memi, i', o' >$
    - $o = o'$ si $I i = i'$
    - $mem'(y) = mem(y)$ pentru orice $y \neq x$
    - $mem'(x) = I$

Instrucțiunea de secvențiere

- $dsemIs : IS \times S \to S \cup \{error\}$
- IS este mulțimea tuturor instrucțiunilor de secvențiere
- În cazul unei liste vide ϵ
  - $dsemIs(\epsilon, s) = s$
- În cazul unei liste $T; L$
  - $dsemIs(T; L, s) = error$
    - dacă $dsem(T, s) = error$; altfel
  - $dsemIs(T; L, s) = dsemIs(L, dsem(T, s))$
  - Dsem descrie semantica instrucțiunii T

Instrucțiunea if

- `if B then L1 else L2 end if`
- b este o expresie care
  - daca $b = 0$ atunci este falsa
  - daca $b \neq 0$ este adevarata
- $L1, L2$ sunt secvențe de instrucțiuni

Instrucțiunea If

- $dsemIf : IF \times S \to S \cup \{error\}$
- IF este mulțimea tuturor instrucțiunilor if
- $dsemIf(ifBthenL1elseL2endif, s) = error$
  - dacă $dsemEx(B, s) = error$; altfel
- $dsemIf(ifBthenL1elseL2endif, s) = dsem(L1, s)$ dacă $dsem(B, s) \neq 0$; altfel $= dsem(L2, s)$

Instrucțiunea While

- `while B do L end while`
- b este o expresie care
  - daca $b = 0$ atunci este falsa
  - daca $b \neq 0$ este adevarata
- L este secvența de instrucțiuni
- $dsemWhile : WHILE \times S \rightarrow S \cup \{error\}$
  - $WHILE$ este mulțimea tuturor instrucțiunilor while

Instrucțiunea While

```
dsemWhile(while B do L end while, s)=error
 daca dsemEx(B,s)=error; altfel

dsemWhile(while B do L end while, s)=s
 daca dsemEx(B,s)=0; altfel

dsemWhile(while B do L end while, s)=error
 daca dsemIS(L,s)=error; altfel

dsemWhile(while B do L end while, s)=dsemIs(L,s)
```

Bibliografie

1. Brian Kernighan, Dennis Ritchie, C Programming Language, second edition, Prentice Hall, 1978.
2. Carlo Ghezzi, Mehdi Jarayeri – Programming Languages, John Wiley, 1987.
3. Horia Ciocarlie – Universul limbajelor de programare, editia 2-a, editura Orizonturi Universitare, Timisoara, 2013.