Facultatea de Calculatoare
Proiect de Diploma

Interpretor si limbaj grafic vectorial
Interfata software/hardware cu un plotter

Student: Sebastian Glita   Coordonator : As.Ing. Sebastian Fuicu

17 iunie 2002

Cuprins

Capitolul 1

Introducere

Acest proiect de diploma este format din patru parti:

- limbaj de programare si (un) interpretor scris in limbajul C pentru el;
- dispozitiv mecanic de desenare intr-un spatiu bidimensional numit *plotter*;
- interfata electronica cu portul paralel al unui PC;
- interfata software intre sistemul de operare Linux si plotter (driver)[1].

Cele patru parti pot coopera impreuna, permitand realizarea unor desene pe hartie cu ajutorul calculatorului. Algoritmii care realizeaza unele desene complexe nu sunt implementati, ci este oferit doar suportul (eficient) ce permite implementarea lor ulterioara.

Proiectul poarta denumirea *plint*, care este o abreviere a sintagmei englezesti *Plotter Language INTerpretor*, care s-ar traduce: Intepretor pentru un limbaj de plotter. Denumirea s-a ales avand in vederea asemanarea cu verbul *p***r***int* din engleza, care se stie ca inseamna a tipari; deci proiectul are ca rezultat asigurarea unui mod de tiparire vectorial, in unele privinte superior celui rasterizat (al unor imprimante clasice):

WITH THIS TOOL, YOU DON'T JUST P*R*INT, BUT **P*L*INT** !

Dispozitivul mecanic si interfata electronica alcatuiesc impreuna dispozitivul electromecanic numit *plotter*, care din punctul de vedere al interpretorului este cunoscut ca plotterul *plint*.

Intre acest dispozitiv si interpretor exista o independenta, in sensul ca interpretorul poate folosi ca "plotter" si alt dispozitiv, cum ar fi o fereastra X[2], si pe de alta parte dispozitivul *plint* poate fi comandata si fara interpretor.

Driverul este de asemenea dezvoltat independent de interpretor, insa dependent de plotter. Exista insa diferite moduri in care se poate realiza comanda finala pe portul paralel, astfel ca exista mai multe tipuri de drivere pentru acelasi plotter.

Legatura dintre interpretor si plotter se face prin intermediul unor fisiere antet/sursa C, numite *primitives.h*, respectiv *primitives.c*. Acestea au rolul de a descrie/implementa colectia de operatii grafice vectoriale *primitive*, precum si unele de control.
Prin urmare, a crea o implementare *primitives.c* inseamna a crea un "plotter", din punct de vedere al interpretorului.

[1] software de control

[2] X este o aplicatia ce realizeaza interfata grafica in sistemele UNIX

Se poate deci spune ca dispozitivul “plotter” e un caz particular al unui “plotter” suportat de interpretor.

Limbajul de programare si interpretorul au fost dezvoltate in paralel, asa incat se confunda, adica se poate spune ca interpretorul determina limbajul. Totusi, limbajul exista ca entitate separata si independenta; se poate descrie formal lexicul, gramatica si semantica acestuia, independent de orice implementare concreta intr-un interpretor/compilator, si oricine poate face un interpretor/compilator pe baza acestor descrieri.
Prin urmare, in continuarea acestei documentatii se va incepe mai intai cu descrierea limbajului, apoi a unei implementari a acestuia, interpretorul *plint*, si pe urma a unui plotter particular, anume *plint*.

1.1 Motivatie

Realizarea unui asemenea proiect mi-a dat posibilitatea sa ajung la cunoasterea unor serii de tehnici din diverse domenii ingineresti — mecanica, electronica, electrotehnica, informatica —, precum si a aplicarii lor impreuna intr-o finalitate logica si functionabila, lucru care stimuleaza creativitatea si ofera satisfactii.

1.2 Limbaj

Limbajul *plint* este un limbaj de programare grafic-vectorial, cu gramatica si semantica extensibila (pluggable [3]).

Este un limbaj de programare, deoarece are notiunile de program, date, intrare/iesire, control.
Este un limbaj grafic-vectorial pentru ca permite “programarea” unor comenzi de desenare bidimensionala vectoriala, adica prin puncte absolute si deplasamente relative.
Are o gramatica extensibila deoarece se pot adauga noi forme gramaticale care extind unele puncte cheie ale gramaticii de baza (core [4]).
Are o semantica extensibila deoarece semnificatia aplicarii unor operatii asupra datelor vehiculate nu este determinata de o anumita descrirere a operatorilor (de exemplu din punct de vedere aritmetic), ci de semnficatia care se atribuie operanzilor, iar operanzii pot avea diverse tipuri; aceste tipuri nu sunt “batute in cuie”, ci ele pot fi adaugate printr-o interfata de functii callback, numita *hook*[5].

1.3 Interpretor

Interpretorul este scris in limbajul C, cu ajutorul generatorului de analizor lexical automat *flex* si a a celui de analizor gramatical automat *bison*.
In principiu, un interpretor executa actiuni sub comanda lexemelor aparute la intrare, fiind citite de obicei dintr-un fisier text, a carui continut poarta numele de cod sursa. De fapt, daca nu ar exista codul sursa *plint*, efectul echivalent s-ar putea obtine si prin scrierea in C a unui program care ar fi foarte lung si dificil de creat si de corectat.
Toate actiunile care par ca fiind efectul “executiei” codului sursa *plint*, sunt de fapt efectul codului interpretorului, care “se lasa” comandat de starile unui automat finit, a carui evolutie in spatiul starilor este determinata in timp de codul sursa *plint*, mai exact de modul

[3] to plug-in = a pune in priza
[4] miez
[5] carlig

de insiruire a lexemelor.

1.4 Plotter

Plotterul este un dispozitiv de desenare plana 2D, comandat de la o placa electronica interfatata pe portul paralel (PP) al unui PC.

Capitolul 2

Limbajul PLINT

Un *limbaj* este de fapt modelarea prin simboluri a unui sistem sau a unui alt model. Atata timp cat pentru cei care vehiculeaza simbolurile acestea reprezinta aceeasi entitate, ei "se pot intelege" in limbajul respectiv. Se stie ca stiinta gramaticii limbajelor de comunicarea dintre oameni este cea care se ocupa cu suprinderea unui manunchi de reguli fixe cu care se pot genera toate frazele posibile. Limbajul este multimea tuturor "lexemelor" — frazelor — posibile, dar de obicei se pot gasi *reguli* intr-un numar fix care pe baza unui *alfabet* sunt suficiente pentru a comprima descrierea **in totalitate** a unui limbaj.

Asemenea limbajului om-om, limbajul om-calculator sau *limbajul de programare* este de fapt o pereche *alfabet-reguli*; de-a lungul istoriei calculatoarelor s-a distins *alfabetul* **ASCII**, care cuprinde simbolurile literelor din alfabetul latin, semnele de punctuatie si ortografie, cifrele arabe si alte inca cateva simboluri. Bazat pe acest alfabet, intr-o serie de limbaje de programare aparute s-au distins de asemenea niste *reguli* mai des folosite, ca rezultat al definirii unor notiuni inerente programarii.
Arhitectura *von Neumann* bunaoara, defineste un sistem de calcul ca avand:

- memorie (de date (MD) si instructiuni (MI))
- unitate aritmeticologico (UAL)
- unitate de control (UC)
- interfata intrare/iesire (IIE)

Nu se specifica nimic despre implementarea concreta a acestor entitati, reprezentand de fapt doar niste *repere*.
Functionarea unei asemenea masini este urmatoarea:
Unitatea de control citeste instructiuni din memorie (operatori, verbe, actiuni), le decodifica (identifica), mai citeste eventual date din memorie (operanzi, substantive, argumente) si dependent de tipul instructiunii le *prelucreaza* cu ajutorul UAL sau le *transfera* prin intermediul IIE. Deci o asemenea masina poate fi vazuta ca un sistem cu intrari si iesiri de date, care *prelucreaza* sau/si *transfera* datele dinspre intrare spre iesire, intr-un mod ce-l face sa fie *folositor*.
Memoria este comuna datelor si instructiunilor, ce reprezinta de fapt o unitate de memorie (byte) depinzand de *interpretare*.

Se poate spune ca sistemul de calcul incearca sa *inteleaga* ceea ce i se comunica, folosind ca *reguli - instructiunile* si ca *alfabet - datele*. Executia unui *program* pe o asemenea masina putem spune ca este asemanatoare rolului pe care il are un *ascultator* in procesul de comunicare cu un *vorbitor*; un program este de fapt o secventa de limbaj. Comunicarea poate avea ca motivatie pe ea insasi, insa de obicei rolul ei este sa determine actiuni din partea *ascultatorului* ca urmare a cererii *vorbitorului*; analog, cererea pe care o *asculta* un sistem de calcul sunt datele de intrare, iar actiunile sunt datele de iesire.

Limbajul uman fiind mai evoluat decat cel masina, este mult mai complex si redundant totodata; pe de alta parte, limita tehnologica a impus ca limbajul masina sa fie cat mai sintetic, pe deasupra folosind ca simboluri numerele 0 si 1, in baza 2; prin urmare, pentru "a exprima" cateva "cuvinte" din limbajul uman in limbaj masina, ca de exemplu:

"Rezolva ecuatia de gradul doi cu parametrii urmatori: a egal cu unu, b egal cu doi, c egal cu unu."

necesita deprinderea de a manipula numere binare si un efort intelectual pentru codificarea in aceste numere a programului (secventa corespunzatoare in limbaj masina):

"`1011010011010101010101011011110111101010101010011100101110101001001 0`"

Intelege cineva ce este scris mai sus ?

"`B4D555BDEAA9CBA92`"

Dar mai sus ? Ba bine ca nu ! Motivul este ca interpretarea lor necesita ca premise existenta unui proces anterior de invatare si experimentare din partea programatorului intr-un *limbaj masina* oarecare; ori, interactivitatea cu calculatorul poate avea de celalalta parte orice persoana umana.

Mai usor ar fi ca masina de calcula sa inteleaga limbajul uman. De aici apare teoria limbajelor de programare de nivel inalt. Acestea sunt de fapt limbaje hibride, o imperechere intre limbajul uman si masina, pentru a usura efortul intelectual al omului si a ramane in limita tehnologica impusa de compromisul complexitate-rapiditate-pret. Cu cat compromisul devine din ce in ce mai flexibil, cu atat limbajele vor fi din ce in ce mai "inalte", tinzand spre vremea cand vom putea vorbi sau chiar gandi impreuna cu calculatorul, direct in limbajul nostru cel mai natural.

Revenind la gramatica unui limbaj de programare, aceasta trebuie sa ofere notiuni corespunzatoare celor patru repere ale unei masini von Neumann. Astfel organizare datelor corespunde notiunii de "structura de date" sau "tip de date", datele efective organizate care se vehiculeaza corespund "variabilelor", modul in care UC se foloseste de UAL in timp pentru a prelucra un set de date corespunde "algoritmilor". In mod normal, cursul unui program este liniar. Pentru a fi posibila implementarea oricarui algoritm, mai este nevoie de posibilitatea de a "sparge" liniaritatea, prin "salturi" spre alte zone de instructiuni; aceste salturi trebuie sa tina cont de valorile instantanee ale datelor, pentru a pastra "generalitatea" algoritmilor, fiind nevoie deci si de "salturi conditionate". Matematic, un algoritm este de fapt o functie definita pe domeniul datelor de intrare, punandu-le in corespondenta datele de iesire. Intr-un proces formal matematic, functiile au un rol doar calitativ. Din perspectiva informaticii, cantitativ ar fi nevoie de o memorie imensa pentru a definii o functie, alaturand fiecarei valori posibile a datelor de intrare, valoarea corespunzatoare din codomeniul functiei. Algoritmii sunt de fapt forma comprimata/codificata a acestei memorii.

Ideal, beneficiind de o memorie infinita, ar fi posibila suprinderea tuturor functiilor, nemaifiind necesari algoritmii; de fapt, in aceasta ipoteza n-ar mai fi necesar timpul, totul fiind "inghetat" in aceasta memorie infinita; in realitate insa, materia se modifica in timp, aceasta modificare putand fi intepretata ca o codificare a unei functii. De exemplu:

- copii care invata sa numere pe degete prin procesul de strangera a pumnului si apoi eliberarea pe rand a fiecarui deget, unul dupa altul, se poate spune ca reprezinta algoritmul de determinare al numerelor naturale de la 0 la 10.
- procesul de dublare al numarului de celule biologice (de un anumit tip) poate fi considerat ca fiind algoritmul asociat functiei exponent in baza 2.
- procesul de oscilatie al curentului printr-un circuit LC acordat reprezinta algoritmul functiei sinus.

Se poate spune ca evolutia in timp a materiei (camp sau substanta) este revelarea entitatilor matematice, acestea fiind adevarata realitate.

Algoritmii complecsi corespund unor functii pentru care analogii ca cele de mai sus sunt mai greu de gasit. Totusi s-a demonstrat ca asigurarea unor premise intr-un sistem de calcul permite executia oricarui algoritm, intr-un timp finit. In cadrul unei masini, o functie (algoritm) este asociata evolutiei unei zone de memorie pe o perioada de timp finita; multimea tuturor bitilor de memorie, pentru fiecare moment discret de timp, intre anumite limite, este de fapt procesul de determinare a iesirii unui algoritm, pe baza datelor de intrare de la inceput; in situatia unei memorii infinite, timpul de determinare ar fi 0.

Anumite parti din aceasta zona de memorie raman nemodificate pe intreaga perioada de calcul, de fiecare data, independent de datele de intrare. Acestea reprezinta zona de program si constante, restul zona de variabile.

Daca in zona de program anumite patternuri se repeta, ele pot fi codificate si stocate o singura data, prin crearea unei conventii de folosire in locul in care se cere: e vorba de notiunea de subrutina, adica blocuri de cod reutilizabile; modalitate de codificare este atribuirea unui "nume" unic. Modul in care ele se utilizeaza poate fi unul prescurtat sau unul generalizat. In primul caz este vorba de situatia cand aceste subrutine accepta intotdeauna unul sau doua argumente ca date de intrare, ele fiind denumite "operatori" unari, respectiv binari, si sintaxa apelului lor face uz de simbolurile aritmetice plus (+), minus (-) etc. In al doilea caz, sintaxa prevede o modalitate de a descrie argumentele intr-o lista oricat de lunga, asociata cu numele (din litere si cifre) al subrutinei apelate; s-a inradacinat modul de apel cu numele functiei in fata (stanga), urmat de o lista separata prin virgule, incadrata intre paranteze rotunde.

Zona de date poate fi privita ca evolutia unei instante a unei structuri abstracte de date; se stie din matematica ca oricarei multimi numarabile i se poate asocia o functie bijectiva numita functie "index", ce pune in corespondenta elemente ale unei multimi index cu cele ale multimii initiale. Pentru programatori, este cunoscuta multimea index a numerelor intregi. O structura de data poate fi "indexata" insa si cu stringuri[1], perechi de numere intregi etc. Limbajul *plint* foloseste notiunea de structura de date (SD) numarabila ce poate fi indexata prin:

- numere intrege (SD numarabila)
- identificatori (SD nominabila)
- siruri de caractere (SD general indexabila)

[1]siruri de caractere ASCII

2.1 Insusiri

Limbajul *plint* se aseamana limbajelor de nivel inalt C, Pascal, Python, precum si shell-scripturilor; asemanarea este atat ca sintaxa, cat si ca semantica:
Se aseamana cu C prin aceea ca blocurile de cod se incadreaza intre {}. Se aseamana cu Pascal prin aceea ca blocurile de cod sunt incuibate, realizandu-se astfel un arbore de asemenea blocuri. Se aseamana cu shell-scripturile prin aceea ca variabilele nu au tipuri statice, ci tipul lor se modifica dinamic.

A vorbi despre limbaj in aceasta sectiune, inseamna a vorbi despre lexic, sintaxa si semantica. Orice executie a unui program sursa *plint* presupune doua faze:

faza de parse in care nu se executa propriu-zis nimic ci doar are loc o verificare sintactica si o constructie a structurilor de date statice din punct de vedere al executiei ulterioare

faza de run in care are loc executia propriu zis a blocurilor, conditionata de date care nu sunt cunoscute decat in acel moment.

Limbajul nu e tipizat static, adica o variabila poatea avea pe parcursul interpretarii mai multe tipuri diferite de variabila, cu tipuri de valori diferite. La un moment data insa, o variabila are un tip, un tip valoare si o instanta a domeniului valoric. Este oferit suport implicit pentru implementarea incapsularii prin folosirea metodelor.

2.2 Lexic

Lexicul limbajului este multimea de lexeme (insiruiri de orice lungime pozitiva de caractere ASCII) ce se poate forma pe baza unor reguli bine precizate (neambigue), plecand de la asa numitele "terminale" — lexeme atomice. Aceste terminale sunt caracterizate de anumite reguli lexicale de construire/recunoastere a lor fixe; o regula simpla ar fi:

Un terminal este o insiruire de lungime si continut fixe de urmatoarele caractere: 'w' 'h' 'i' 'l' 'e'.

O alta ar putea fi:

Un terminal este o insiruire de lungime variabila si continut litere sau cifre, dar pe prima pozitia numai litere.

Aceste terminale sunt identificatorii din limbajele de programare.

Tinand cont de aspectele teoretice ale analizei lexicale, s-au stabilit niste tipuri de reguli numite "expresii regulate", cu care se pot descrie lexemele oricarui limbaj. Expresiile regulate sunt un metalimbaj pentru descrierea limbajelor; ele ar putea folosi ca si alfabet simboluri extracalculator, insa nu ar folosi la nimic. Existenta acestor simboluri ar fi insemnat coduri in plus pe aceeasi pozitie (deci un fel de comprimare); solutia alternativa e folosirea setului ASCII pentru simboluri, si mai multe pozitii pentru extrasimboluri. De exemplu, '`w`' inseamna dublu-v, insa '`\w`' inseamna "orice secvente formata din orice caractere, in afara de litere, mari sau mici". In teoria expresiilor regulate, descrierea lor se face folosind niste terminale speciale, numite "metacaractere"; acestea sunt simboluri ASCII, cu interpretare diferita decat ce triviala; pentru a le folosi cu semnificatia lor originala, ele trebuiesc precedate de un alt metacaracter si mai special, anume '\' (backslash). Orice metacaracter precedat de '\' isi pierde semnificatia de metacaracter, devenind trivial; backslashul cel precede este ignorat. Din cele spuse anterior rezulta ca '\\' e de fapt backslashul trivial.

Lexicul este format din mai multe clase lexicale:

- spatii albe (whitespaces): SPACE NEWLINE TAB
- comentarii /* ignore me */
- operatori : +=, =, &&
- punctuatori : ;, ,, , , [,]
- specificatori: $ " @ # !
- reprezentari literale ale valorilor, cum ar fi numere, siruri de caractere etc.
- identificatori
- cuvinte cheie (keywords)
- functii (functions)
- combinatii intre clase

2.2.1 Spatiile albe

In general, pe o foaie scrisa se disting "spatiile albe", adica acele portinui nescrise. Acest "spatiu alb" ar putea sa-i atribuim un simbol generic; reprezentarea grafica a lui e imposibila insa, depinzand de densitatea "spatiului negru". De aceea, el se imparte in:

newline pe fiecare linie, tot "spatiul alb" pana de la sfarsit pana la marginea din drepata

space cel mai mic interval "alb" intre doua cuvinte

tab un interval "alb" de lungime fixa si repetabila

Spatiile albe nu fac parte din alte terminale, putand fi folosite la separarea celorlalte.

2.2.2 Comentarii

Comentarii sunt tot un fel de spatiu alb, daca nu fizic, atunci sigur informatic, adica informatia ce o contin se ignora. Comentarii pot fi pe linie, sau pe bloc; in primul caz se ignora tot ce urmeaza dupa `\\`, pana la sfarsitul liniei; in al doilea caz se ignora toate caracterele (posibil pe mai multe linii) intre `/*` si `*/`.

2.2.3 Operatori

Operatorii sunt simboluri ce se pot folosi in formarea expresiilor de calcul; o expresie de calcul realizeaza o serie de operatii asupra unor operanzi, folosind operatorii dintre ei. Operatorii au precedenta si asociativitate; precedenta spune in ce ordine trebuie executate operatii care altfel ar putea fi executate in orice ordine; odata stabilit acel operator care are precedenta cea mai mare, daca se intalnesc mai multi asemenea consecutivi (deci cu precedenta egala), asociativitatea de la stanga la dreapta sau invers determina un alt fel de precedenta ce stabileste ordinea executiei.

2.2.4 Punctuatori

Punctuatorii sunt folositi pentru a schimba regula de executarea a operatiilor dupa precedenta si asociativitatea lor, precum si ca delimitatori intre diferite blocuri, apeluri de functii, primitive, instructiuni etc.

2.2.5 Literale

Literalele sunt lexeme speciale din care se poate construi *direct* o valoare: de exemplu, o insiruire de cifre este un numar intreg. Nu toate valorile au literale; in acest caz valorile pot fi obtinute *indirect* (oarecum literal) folosind operatori si/sau (numai si ???) punctuatori.

2.2.6 Identificatori

In afara de clasele amintite mai sus, orice insiruire de litere[2] si cifre, avand pe prima pozitie obligatoriu o litera, intra in clasa identificatorilor. Acestia pot fi nume de variabile[3].

2.2.7 Cuvinte cheie

Cuvintele cheie sunt identificatori care au clasa lor proprie, formata dintr-o singura lexema. Un cuvant cheie este deci o clasa de identificatori, cu un singur element, insusi identificatorul cuvantului cheie. In aceasta acceptiune, tot restul identificatorilor intra intr-o singura clasa, anume clasa identificatorilor (triviali).

2.2.8 Functii

Functiile sunt cuvinte cheie pentru care se face o verficare in plus, a lexemei ce urmeaza lui, anume poate fi spatiu alb urmat neaparat de paranteza rotunda deschisa.

2.2.9 Specificatori

Uneori, unele constructii de limbaj necesita intr-o anumita pozitie valori de un anumit tip (de exemplu, argumentul numeric numit "raza cercului" la apelul primitivei grafice "circle"). Reprezentarile literale (ale numerelor in acest exemplu), pot aparea fara probleme; daca se doreste valoarea unei variabile numerice insa, simplul identificator nu ar putea fi de ajuns, deoarece exista cazuri cand ar duce la ambiguitate (de exemlu, argumentele de tip punct de la primitivele grafice pot fi puncte sau o lista de 2 numerel; pentru a distinge pe ce "cale" se merge, e nevoie de la bun inceput sa se distinga cele 2 cazuri; daca argumentele ar incepe cu valori literale ale punctelor sau numerelor, acestea se disting prin definite, deci n-ar fi nici o problema; daca insa se da numele (identificatorul) unei variabile, automatul gramatical generat de *bison* se va afla intr-o stare nedeterminista, "nestiind" ce sa faca: e vorba de o valoare punct sau de un numar ? Asta deoarece analizorul gramatical nu "stie" nimic de semantica). Specificatorii rezolva aceasta problema, fiind niste caractere ce prefixeaza identificatorii variabilelor, atunci cand nu se poate altfel (anumite constructii gramaticale accepta identificatori fara specificatori).

2.3 Sintaxa

Lexicul limbajului este multimea de lexeme (insiruiri de orice lungime pozitiva de caractere ASCII) ce se poate forma pe baza unor reguli bine precizate (neambigue), plecand de la asa numitele "terminale" — lexeme atomice. Asa cum o ecuatie diferentiala are mai multe solutii functie de conditia initiala, la fel o secventa de limbaj poate fi redusa la un sir de terminale in mai multe moduri, devenind ambigua. Detectarea tuturor acestor posibilitati este o pierdere de timp, deoarece se stie ca omul e obisnuit "sa citeasca" o fraza de la

[2] Prin litera se inteleg literel romane a-z, A-Z si _

[3] Chiar si de valori: 'true', 'false'

inceput la sfarsit, si nu de la mijloc sau de la trei-sfert. De aceea, punand “conditia initiala” ca analiza/recunoasterea lexicala sa inceapa de la inceputul unei secvente de limbaj, se reduce efortul de construire a unei gramatici neambigue din acest punct de vedere. Iata un exemplu: presupunem ca avem un limbaj cu gramatica urmatoare:

$\langle x \rangle$ ::= $\langle i \rangle$ $\langle j \rangle$ | $\langle k \rangle$ $\langle l \rangle$

$\langle i \rangle$ ::= ‘`a`’

$\langle j \rangle$ ::= ‘`b`’ ‘`a`’

$\langle k \rangle$ ::= ‘`a`’ ‘`b`’

$\langle l \rangle$ ::= ‘`a`’

Rezultatul este un automat ce se poate vedea in anexa 6.4. Daca se incearca analizarea sirului “aba”, se observa ca el poate fi vazut fie ca “a” urmat de “ba”, fie ca “ab” urmat de “a”. dupa cum se merge pe calea ‘ij’ sau ‘kl’. Generatorul *bison*, dupa cum se vede din iesirea descriptiva a acestuia, a detectat un conflict care apare dupa citirea primului caracter ‘a’, automatul creat aflandu-se intr-o situatie nedeterminista; in acecst caz va lua o decizie implicita, urmand calea ‘ij’.

Terminalele pot fi caractere, siruri contiguue de caractere (o regula liniara de detectie a lor), sau coduri asociate unor constructii lexicale mai complicate (posibil expresii regulate). Terminalele, ca si neterminalele, sunt unice. Se poate defini ca terminal numai unitatea “caracter”, gramatica analoaga ar fi insa foarte foarte dificil de conceput; de aceea s-au permis terminale mai complexe, beneficiind de reutilizare prin reguli lexicale si coduri.

Neterminalele sunt entitati strict gramaticale, ele fiind niste identificatori [4] asociati [5] unor reguli, posibil alternative, de succesiune a altor neterminale sau terminale. Se considera ca o lexema formeaza un neterminal, daca regula asociata acestuia din urma se aplica corect pentru fiecare element (terminal/neterminal), de la inceputul pana la sfarsitul lexemei. Comentariu poate

2.3.1 Antet

Antetul este o secventa de lexeme ce se supun unor reguli gramaticale care nu mai apar nici unde altundeva; este vorba de:

autorun Este o facilitate ce permite shellurilor sa lanseze indirect aplicatia plint, prin specificarea unei secvente text ce incepe cu “!” si e urmata de calea interpretorului fisierului. Pentru a nu se incurca cu limbajul propriu-zis, a fost permisa la inceeput posibilitatea exitentei unei linii de acest fel.

versiune Specificarea versiunii cu care a fost testat bine ultima data fisierul sursa *plint*, permite determinarea la un moment de rulare ulterior daca versiunea interpretorului curenta e compatibila cu cea anterioara.

plottere Anumit tipuri de plottere pot fi incompatibile cu unele aspecte ale limbajului, de aceea daca asemenea aspecte se regasesc in fisierul sursa, atunci trebuie specificat lista cu plotterele compatibile.

[4] atentie, nu in intelesul lexical, ca siruri de litere, cifre

[5] intr-un fisier de descriere gramaticala, cu o sintaxa proprie generatorului folosit

2.3.2 Program principal

Orice program sursa plint, pentru a fi rulat, trebuie sa contina un program principal. Acesta este format dintr-un nume[6] si blocul principal. Programul principal este o procedura si contine o variabila readonly numita "argv", care este un tablou de argumente. Este deci posibil apelul recursiv al procedurii principale doar daca numele ei este nume de identificator, pentru ca in alt loc decat aici nu se mai permite denumirea/apelul blocurilor cu nume altfel decat de identificatori.

2.3.3 Perspectiva statica

Perspectiva statica urmareste modul de dispunere a terminalelor si neterminalelor in programul sursa.

Structura

Structura unui program sursa *plint* este un arbore cu nodurile blocuri de cod; aceste blocuri pot fi de diferite tipuri; exista 3 tipuri de blocuri standard:

anonime sintaxa ca si in C, blocul este incadrat de {}

nominale acestea sunt procedurile, functiile, figurile, metodele.

infasuratoare blocuri "transparente"; ajuta la crearea unor functionalitai mai avansate cum ar fi iteratii for, while etc.

Inserare Mai exista un tip de noduri in arbore, anume nodurile de insertie; acestea sunt transparente atat din punct de vedere dinamic cat si static; transparenta statica se refera la faptul ca nodurile fii ale unui nod de insertie sunt noduri fii (virtual) si ale nodului lor parinte, precum si de faptul ca diferenta dintre nivelurile a doua noduri pe un drum in arbore, intre care sunt numai noduri de insertie sau nici unul este 1; transparenta dinamica se refera la faptul ca drumul spre parinte al unui nod bloc poate diferi daca deasupra lui se afla cel putin un nod de insertie.

Pentru inserarea sau includerea altor fisiere cu cod sursa *plint* se foloseste sintagma:

`insert 'nume/cale fisier'`
sau
`include 'nume/cale fisier'`

Diferenta intre cele doua este ca, in ultimul caz, daca fisierul a fost deja inserat sau inclus, el nu se mai insereaza o data, ci numai procedurile se fac disponibile in blocul curent.

Nivelul unui bloc este dat de adancimea lui in arbore, in considerentele de mai sus.

Blocuri normale Blocurile normale ajuta la gruparea unei zone de cod in care, de exemplu, se pot folosi nume de variabile la fel cu altele de pe un nivel mai sus, fara a avea conflict de nume.

[6] nu neaparat un identificator!

Proceduri, Metode Procedurile si metodele se deosebesc de primitive prin aceea ca sunt construite in cadrul limbajului *plint*, pe cand primitivele sunt doar “porti” de apel a unor functii C; sintaxa de apel se aseamana. De asemenea, procedurile/metodele intotdeauna returneaza o valoare, pe cand primitivele sunt pasive din acest punct de vedere. Iata un exemplu:

```
center = [0, 0];
moveto (@center);

procedure _lineto (x)
{
  moveto (@center); // primitiva
  lineto (@x); // primitiva
}

new x, xx;
x = @center + [10, 10];

// now you can call
_lineto (@x); // procedura

// or
_lineto (x);
```

Vizibilitate Vizibilitatea blocurilor nominale este astfel:

- sunt vizibile toate blocurile nominale de pe nivelele superioare gasite in drumul spre radacina arborelui de blocuri
- sunt vizibile toate blocurile nominale de pe acelasi nivel cu blocul curent, sau cu cele care respecta conditia de mai sus

2.3.4 Perspectiva dinamica

Perspectiva dinamica urmareste modul in care aceleasi terminale si neterminale evolueaza in timpul rularii programului sursa.

Nu exista nici o diferenta intre proceduri, functii, figuri; codul terminalului corespunzator cuvintelor cheie “procedure”, “function”, “figure” fiind acelasi. In continuare referirea la proceduri este echivalenta cu referirea sau la procedurim, sau la functii, sau la figuri. Cand valoarea returnata de o procedura e folosita, procedura functioneaza ca functie; cand comenzi de desenare se aduna in cadrul unei proceduri, avem de-a face cu o figura; scopul existentei mai multor variante de denumire a procedurilor este usurarea intelegerii codului sursa *plint*.

Metodele difera de proceduri prin aceea ca ele trebuie apelate mai special, folosind operatorul de apel de metoda “’, aplicat unei referinte; de aceea ele au ca argument ascuns o variabila special numita *self*.

Numele procedurii curente pote fi obtinut din variabila *this*; prin *this* se si poate apela recursiv procedura curenta.

Fluxul

Fluxul programului este o succesiune de stari ale automatului analizor gramatical si ale stivei de blocuri. Fiecare bloc al programului poate avea mai multe instante pe stiva in timpul rularii. Modul in care se sare dintr-un bloc in altul este dat de o regula simpla: - in blocurile normale se intra neconditionat - procedurile de pe acelasi nivel sau mai mare se pot apela prin numele lor

Din blocuri se poate iesi fortat prin folosirea instructiunii de control “break n” sau “continue n” si se poate bucla prin instructiunea “continue loop”.

Fiecare bloc are o variabila implicita, numita “_” care poate fi folosita cand nu e nevoie de sa se declare neaparat o noua variabila. Variabila blocului de pe un nivel mai inalt, fara a trecec peste procedura curenta, se poate referi precedand simbolul “_” cu diferenta de nivel corespunzatoare, maxim 9. Astfel, “0_” e echivalent cu “_”, “1_” cu “_” din blocul care gazduieste direct blocul curent etc. Exemplu:

```
  {
  loop (2)
  {
    loop (3)
    {
    loop (4)
    {
      loop (5)
      {
      loop (6)
      {
        loop (7)
        {
        loop (8)
        {
          loop (9)
          {
          loop (10)
          {
echo 0_,'␣',1_,'␣',2_,'␣',3_,'␣',4_,'␣',5_,'␣',6_,'␣',7_,'␣',8_,'␣',9_,;
          }
          }
        }
        }
      }
      }
    }
    }
  }
  }
```

Rezultatul va afisa 1*2*3*4*5*6*7*8*9*10 linii in modul urmator: 10 9 8 7 6 5 4 3 2 1 9 9 8 7 6 5 4 3 2 1 ... 1 1 1 1 1 1 1 1 1 1

Exista instructiuni de control pentru iesirea din procedura curenta; acestea sunt “exit”, “return n” si “escape n”. Prima iese din program; a doua permite iesirea din a n-a instanta a procedurii curente pe stiva; a treia permite iesirea din toate instantele pe stiva ale procedurilor care sunt imbricate pana la un anumit nivel in tinta instructiunii “escape”, calculata ca procedura la un nivel mai maare cu n fata de cea curenta. Valorile implicite pentru n sunt 1 pentru “return” si 0 pentru “escape”.

2.3.5 Perspectiva neutra

Continutul blocurilor Continutul blocurilor nu e ceva ce se poate preciza cu exactitate; poate fi orice sau nimic; tot ce se intampla in executia unui program are loc in cadrul blocurilor, deci putem spune ca blocul atrage toate actiunile in interiorul lui. Aceasta ne poate duce cu gandul la gaurile negre — *blackhole*s— si la ceea ce sunt ele, adica densitate de materie. Continutul unui bloc este densitate de materie, deci un *blackhole*.

Revenind in domeniul concret al limbajelor, o *blackhole* este alcatuita din sectiuni, iarasi o precizare destul de abtracta:

$\langle blackhole\rangle$ ::= $\langle empty\rangle$ | $\langle blackhole\rangle$ $\langle section\rangle$

Sectiunile sunt o colectie de reguli gramaticale grupate dintr-un anumit punct de vedere; am discriminat urmatoarele sectiuni fundamentale[7]:

primitives

structure

data

flow

io

Fiind un limbaj grafic, exista o dimensiune a limbajului care-l separa de celelalte limbaje non-grafice de programare, anume primitivele grafice. Acestea sunt niste reguli gramaticale care nu au legatura cu caracteristica de limbaj de programare, ele fiind un fel de "capsuna de pe varful prajiturii".

Din puncte de vedere static, un program este un arbore de blocuri; acestea provin din fie direct din fisierul sursa, fie din alte fisiere incluse. Structura este deci colectia de reguli privitoare la blocuri si fisiere; aceste reguli se vor reduce doar in prima faza a interpretarii. Aceste reguli pot implica si manifestarea activa a conceptului de "data", in acest caz e vorba de date *statice*.

Din puncte de vedere dinamic, majoritatea regulilor prevazute in sintaxa limbajelor se tot reduc; daca nu ar exista posibilitatea iteratiilor sau apelurilor recursive, executia ar fi liniara; la un moment dat insa, la terminarea executiei unui bloc, se poate relua executia de la inceputul lui - iteratii. Tot asa, intr-o procedura se poate apela pe ea insasi direct sau indirect, ceea ce duce la spargerea liniaritatii.

Aceste reguli care se tot reduc pot fi tangente ca semnificatie cu conceptul de "date", "flow" sau "io". Cand e vorba de date, e vorba despre variabile *dinamice* (echivalentul atributului "auto" din C). Cand e vorba de flux, este implicat controlul asupra punctelor de intrare/iesire din blocurile de cod, precum si de facilitatea de abandon ("abort") a tuturor instantelor blocurilor pana la un o anumita instanta a unui alt bloc (daca e una intermediara, e cazul instructiunii "return"; daca e explicit "ultima", e cazul instructiunii "escape"). Ca limbaj grafic, nu ar fi necesara partea de intrare/iesire, poate doar ca logger (iesire). Totusi, tinand cont ca s-a dezvoltat pe SO Linux, exista bine definita manipularea canalului standard de intrare, respectiv iesire. Aceasta ofera un nivel de interactivitate ce adauga un plus de atractivitate programarii in acest limbaj.

[7] cam seamana cu descrierea arhitecturii von Neumann, nu ?

2.4 Semantica

Semantica unui limbaj este ceea ce conteaza in final; lexicul si sintacticul sunt o modalitate usoara si precisa de a forma un canal neperturbat de comunicatie intre om si calculator; efectul comunicatiei este semantica.

Semantica unei primitive grafice este de a fi calea de acces spre dispozitivul grafic de "jos"; tipul primitivei determina modul de interactiune, prin lista de argumente.

Semantica unei directive de insertie/includere este de a indica interpretorului sa integreze in arborele static de blocuri "tot" ce gaseste in fisierul argument.

Semantica unui apel de procedura/metoda este de a ramifica executia programului.

Semantica unei instructiuni de control este de a corecta executia programului.

Semantica unei declarari de variabila este de a determina interpretorul sa incerce creearea unui nou slot in lista de variabile, astfel incat sa nu existe doua nume de variabile identice.

Semantica unei atribuiri este de a oferi o "metoda" bine pusa la punct pentru modificarea dinamica a datelor; altfel totul ar fi "inghetat"

Semantica constructiilor literale este de a se transforma intr-o valoare, al carei tip depinde de sintaxa literalului.

Semantica unei operatii intre doua valori sau asupra unei valori este de a creea o noua valoare, dependent de tipul operatiei si operanzilor.

Semantica unei referinte este de a returna "adresa" unei variabile.

Semantica unei variabile este de a returna un pointer la o variabila.

Semantica existentei unui operator unar "langa" o variabila, sau a unui binar "intre" 2 variabile este de a determina o "actiune" dependenta de tipul variabilei, sau respectiv variabilelor.

Semantica unei liste de insantieri de variabila, a unor serii de atribuiri, a unei serii de aplicari de operatori pe un set de variabile este de a efectua mai multe "iteratii" in sensul ramificarii programaului iterativ, pana la parcurgerea si aplicarea unei "operatii" asupra fiecarui element din lista.

2.5 Extensii

2.5.1 Intrare/Iesire

Extensia "io" are rolul asigurarii unei modalitati ce ofera programatorului sa intervina asupra modului de formatare a valorilor la iesire.
Mai ofera o modalitate de suspendare a executiei pana la apasarea unei taste, cu afisarea unui prompt.

2.5.2 Necesar

Toate variabilele au un tip; in miezul limbajului nu exista nici un tip definit pentru variabile; exista insa un tip utilizat virtual, despre care nu se stie nimic mai mult. Prin urmare, aceasta extensie este ceea care ofera functionalitatea tipului virtual din miezul limbajului. Mai ofera o sintagma de creare a variabilelor de acest tip, mai multe deodata.

2.5.3 Necesar

Aceasta este o extensie interesanta; ofera posibilitatea utilizarii variabilelor statice, adica care exista o singura data pentru toate instantele blocurilor de cod. Mai ofera si o modalitatea de accesare a oricarei variabila statica *din orice bloc*, chiar daca nu e vizibil.

2.5.4 Functii matematice

In prelucrarea coordonatelor punctelor sunt utile o serie de operatii matematice, cum ar fi: $\sin, \cos, \exp$ etc. De asemenea implementeaza operatii de determinare a modulului, unghiului, catului unui numar complex, identic ca functionare unui punct 2D.

2.5.5 Rendare de text “freetype”

“Freetype” este o bibilioteca C ce ofera posibilitatea aplicatiilor de a renda tipografic un text in font TrueType. La extremitatea functionalitatii, bibilioteca foloseste 4 tipuri de primitive grafice: moveto, lineto, conicto si cubicto. Ultimele doua sunt curbe bezier cu 3, respectiv 4 puncte, sau altfel spus, cu 2, respectiv 3 puncte de *control*. Este o intamplare fericita ca limbajul *plint* se potriveste foarte bine cu cerintele bibliotecii din punct de vedere al acestor primitive.

Un text se rendeaza folosind “write”. Fontul se alege dand ca argument o cale spre un fisier cu fonturi TrueType (de obicei cele cu extensia .ttf) primitivei “font”. Se poate alege dimensiunea in puncte a fontului, precum si orientarea textului, explicit la fiecare apel “write”, sau implicit, prin primitiva “font”. De exemplu, pagina titlu a acestei documentatii a fost obtinuta cu urmatorul program:

```
#!/usr/bin/plint

plint 'diploma'
{
//    unit (1, 'pt');

    procedure newline (s) {
  moverel (0,pnty(P2(bbox (s)) - P1(bbox(s))));
    }


    /* header */
    moveto (0,10*maxy/100);
    linerel (maxx, 0);



    /* sus */
    align ('center');
    moveto (maxx/2,20*maxy/100);

    font (''argv[1]);

    font ([24,24]);
    _ = 'Facultatea de Calculatoare';
    write (_);
    newline (_);
    font ([20,20]);
    _ = 'Proiect de Diploma';
    write (_);
```

```
    /* mijloc */
    align ('center');
    moveto (maxx/2,50*maxy/100);

    font ("argv[2]);
    font ([16, 16]);

    _ = 'Interpretor si limbaj grafic vectorial';
    write (_);
    newline (_);
    _ = 'Interfata software/hardware cu un plotter';
    write (_);


    /* jos */
    align ('right');
    moveto (99*maxx/100,80*maxy/100);

    font ("argv[3]);
    font ([18,14]);

    _ = 'Student: Sebastian Glita';
    write (_);
    newline (_);
    _ = 'Coordonator : As.Ing. Sebastian Fuicu';
    write (_);


    /* footer */
    moveto (0,90*maxy/100);
    linerel (maxx, 0);

    align ('center');
    moveto (maxx/2,95*maxy/100);

    font ("argv[4]);
    font ([20,12]);

    _ = 'Timisoara, 2002';
    write (_);


    flush ();
    wait 'done.';

}
```

cu urmatoarea comanda:

```
plint title \"../share/fonts/TIMES.ttf \"ARINCE.ttf \"../share/fonts/
   ariosob.ttf \"../share/fonts/greek.ttf
```

2.5.6 Interfata sistem

Se pot afla diferite informatii din sistem, cum ar fi tipul sistemului/platformei, identificatorii de proces propriu si al parintelui, timpul si data curenta etc. De asemenea se pot lansa procese prin intermediul apelului din biblioteca C "system".

2.5.7 Unitati

Faciliteaza setarea indirecta de rezolutii ale plotterului, prin specificarea unitatii dorite si in ce cantitate, astfel incat *distanta* intre 2 coordonate consecutive ale punctelor sa fie egala cu ce s-a setat.

2.5.8 Constructia "if-else"

Introduce instructiunea clasica de conditionare a executiei fluxului programului; O constructie "if-else" incepe cu "if" urmata de o valoare booleana testata, intre paranteze rotunde. Pot urma apoi zero sau mai multe variante "else if"; in final mai poate apare un "else" neurmat de "if". Prima oara cand se intalneste o conditie adevarata in primul "if" sau in "else if"-urile intermediare, sau cand se "ajunge" la ultimul "else" fara "if", se executa blocul corespunzator, iar cele care urmeaza eventual sunt ignorate. Exemplu:

```
if (true) { echo 'ok'; }
else { echo 'bug'; }

if (true) { echo 'ok'; }
else if (false) { echo 'bug'; }
else { echo 'bug'; }

if (false) { echo 'bug'; }
else if (false) { echo 'bug'; }
else { echo 'ok'; }

if (false) { echo 'bug'; }
else if (true) { echo 'ok'; }
else { echo 'bug'; }

if (false) { echo 'bug'; }
else { echo 'ok'; }
```

2.5.9 Constructia "case"

Extinde instructiunea 'if-else'; o constructie 'case' are un argument un numar, punct sau string; aceasta deoarece acestea sunt singurele valori scalare care au numar de valori care nu este eficient a le testa in constructii "if-else" (cum este, de exemplu, valoarea booleana - true sau false). In continuare, pentru fiecare tip de "case" (functie de tipul valorii primului argument), exista variante de potrivire, separate prin ','; lista se incheie cu ';'. Spre deosebire de blocul "if-else", daca s-a potrivit o ramura, se mai pot potrivi si urmatoarele. Daca se doreste o functionalitate asemanatoare "switch-case"-ului din C, se poate incadra tot blocul "case" intr-un bloc normal, si apoi din blocurile de pe ramuri sa se exectue "break;". De exemplu:

```
#!/usr/bin/plint
plint 'interactive range' {
    format string = '','';
```

```
    case ($({echo 'case_val = '; read integer;})) {
  -20,0 : 0,+20 ; { echo 'intre -20 si +20',; }
  -10,0 : 0,+10 ; { echo 'intre -10 si +10',; }
  $({ echo '['; read integer; echo ','; }), $({ read integer; echo
       ']'; }) ; { echo ,'your choice',; }
  else { echo 'else'; }
    }

    echo 'done',;
}
```

2.5.10 Maparea operator-string

Este utila pentru a degreva programatorul de cunoasterea unor reprezentari ca string ale operatorilor, in cadrul procesului de cautare (seek) a unei metode pentru tratarea unui operator.

2.5.11 Tipul "class"

Introduce programarea pe obiecte in *plint*. De fapt in limbajul de baza exista notiunea de *metoda* si *operatori*, precum si de tipuri dinamice. O clasa este o variabila si un tip dinamic totodata. Tipul variabilei este "class"; cand se creeaza o asemenea variabila, ea se seteaza a fi singleton, si toate operatiile posibile cu tipul "class" sunt invalidate. Practic este o variabila "schilodita", a carei singura menire este mentinerea unor tabele de mapari intre stringuri de operatori si nume de implementari de operatori (metode), intre nume de metode membru si nume de implementari de metode, precum si intre nume de date si o valoare implicita pentru toate "obiectele clasei".

Odata cu crearea variabile de tip "class", codul C al pluginului profita de ocazie si cunoscand numele variabilei, inregistreaza un nou tip, care este de fapt clasa obiectelor ce se vor instantia.

O clasa poate avea o clasa de baza, in care caz orice metoda membru a clasei de baza, care nu e ascunsa prin denumire unei metode omonime in clasa derivata, se poate apela prin obiectele clasei de baza, fara efort suplimentar. Practic, din perspectiva C++, *toate metodele sunt virtuale*. Un obiect are in timpul executiei o clasa curenta, si o clasa originala. Initial cele doua coincid; este posibil insa ca pe parcurs sa se faca "cast" la o clasa de baza, sau chiar din alta ierarhie. In acest moment, clasa curenta difera de cea originala; cu toate acestea orice apel de metoda se face din perspectiva clasei originale; in contrast, orice referire la date membru se face din perspectiva clasei curente. Pentru a apela nevirtual metodele clasei curente, se poate folosi data membru "self", care este un obiect alias ce are ca si clasa originala clasa curenta a obiectului container. La fel si operatorii, ca si metodele unei clase sunt virtuali.

Orice obiect instanta a unei clase de baza derivata are o data membru numita "base" care este un obiect.

Datele membru "self" si "base" sunt variabile readonly, a caror clasa curent si orginala coincid totdeauna; ele sunt obiecte alias la obiectul container.

2.5.12 Iteratia "loop"

Ofera posibilitatea iterarii unui bloc de un numar fix de ori, cu un contor implicit descrescator.

```
  if ($_ == 1) { echo 'second;␣}
  if ($_ == 2) { echo 'first'; }
}
```

2.5.13 Iteratia "while"

Ofera posibolitatea iterarii unui bloc conditionat de rezultatul boolean al rularii la fiecare iteratie a unui bloc de conditie. Acesta din urma se include intre paranteze rotunde imediat dupa cuvantul cheie "while"[8].

2.5.14 Iteratia "for"

Implementeaza o functionalitate asemanatoare bine cunoscutei constructii omonime din C. In esenta dupa cuvantul cheie urmeaza trei blocuri separate intre ele prin ':' si de blocul exterior prin '(' ')'. Dupa ')' urmeaza blocul propriu zis al "for"-ului. Dintre cele trei blocuri, primul este de initializare, al doilea este cel conditional, iar ultimul se executa dupa executia blocului propriu-zis, care urmeaza rularii blocului conditional, daca valoarea returnata de acesta, ca boolean, este adevarta. Variabilele din cele 3 blocuri sunt locale si nu se pot folosi in alte blocuri. Exemplu:

```
n = rnd (100);
for (i = 0; : i < n; : ++i; ) {
  echo rnd (i),;
}
```

2.5.15 Iteratia "foreach"

Tipurile de valori care sunt structuri de variabile, cum ar fi tabloul sau hashul, sunt prin definitie iterabile; de aceea aceasta iteratie are un anumit tip de regula si implementare pentru fiecare tip de valoare structurata, primeste ca argument o astfel de valoare si ruleaza un bloc in care valoarea variabilei '_' la intrare este un tablou de 2 elemente, primul indice elementului curent, iar al doilea este chiar elementul curent. Indicele este o variabila a carei valoare are un tip dependent de tipul structurii de variabile pe care o indexeaza; astfel, in cazul tablourilor este un intreg, iar in cazul hashurilor este un string.

[8] Atentie! Probabil veti uita ';' inainte de paranteza ')'! Nu uitati ca este un bloc acolo!

Capitolul 3

Intepretorul PLINT

Un interpretor se poate analiza din perspectiva asemanarilor si deosebirilor fata de un compilator. Pana la un anumit punct, cele doua se aseamana, adica au in comun faza de analiza lexicala si sintactica; de aici insa cele doua incep sa se deosebeasca.
Un compilator bunaoara in momentul detectarii indeplinirii unei reguli, se foloseste de simbolurile corespunzatoare regulii respective pentru a genera o secventa de cod intr-un limbaj de nivel mai jos. Deci interpretarea semantica consta in *generarea de cod.*
Un interpretor, in aceleasi circumstante, va efectua o *actiune* ce este de fapt un cod C ce ruleaza in cadrul interpretorului, deviind firul principal al programului acestuia. Prin urmare un interpretor se *erijeaza* temporar in programul sursa interpretat, asa incat interpretorul si programul sursa sunt unul si acelasi lucru. Mai mult, timpul de rulare al “procesului” programului sursa este inclus in cel de rulare al procesului intepretorului; spre deosebire, programul compilat de un compilator ruleaza intr-un proces separat, intr-un moment de timp ulterior terminarii procesului compilatorului.

Interpretorul *plint* este un program de sine statator, concretizat intr-un fisier executabil cu acelasi nume. Pentru a ajunge in aceasta forma, codul sursa al interpretorului trece prin mai multe faze. Codul sursa este in principal cod C, insa sunt folosite niste “instrumente” software care au fisiere text de intrare cu sintaxa proprie.
Linux este un sistem de operare destinat programatorului, si prevede in mod natural instalarea unor tooluri de analiza lexicala si/sau gramatica. Astfel pentru descrierea lexicului este folosit toolul *flex*, un analizor lexical bazat pe un fisier de descriere a formelor lexicale ca *expresii regulate* gramaticii, pentru descrierea gramaticii se foloseste aplicatia *bison*, un analizor gramatical si generator de actiuni semantice. Rezultatul acestor programe este crearea unor fisiere sursa C care pot fi compilate impreuna cu restul aplicatiei; aceasta integrare este realizat automat sub controlul toolului *make*, un program de construire de aplicatii prin recompilarea doar acelor module care se schimba, in decursulu procesului de dezvoltare. Se mai folosesc *shell script*urille pentru o serie de prelucrari de fisiere text intermediare.

Toate fisierele se gasesc intr-un director oarecare in cadrul directorului radacina. Relativ la respectivul director, exista o ierarhie de directoare proprie aplicatiei, organizata intr-un mod care sa faca cat mai utila parcurgerea lui, precum si din punct de vedere logic, urmarind separarea entitatilor care sunt oarecum independente.

Limbajul fiind extensibil sintactic si semantic, suportul acestei insusiri consta in diferite fisiere sursa C, de lexic, sintaxa, construire si shell script. Afara de acestea, restul fisierelor constituie miezul aplicatie, adica acele fisiere esentiale, care dau o functionare si reprezentare minimala, necesara si suficienta (nu chiar suficienta, vezi extensia “blitin”). Prin urmare, in continuare se va vorbi separat, pe rand, despre miez, extensii si modul cum se integreaza impreuna pentru a rezulta aplicatia finala *plint*.

3.1 Folosire

Rularea programului *plint* din linia de comanda a unui shell, fara argumente sau cu argumentul lung "- -help", va scoate la iesire urmatorul text:

```
plint: usage:
plint [[OPTIONS] file1 [FILE1ARGV] [[OPTIONS|,] file2 [FILE2ARGV]]... ]...

        OPTIONS can be one of:
                -v         : toggles verbose mode
                -p         : toggles parse only (syntax check) mode
                -P         : toggles procedures cache enable/disable
                -V         : toggles variables cache enable/disable
                --plugins  : plugins compiled in
                --plotter  : plotter device used
                --stamp    : machine & time stamp
                --insdir   : insert (include) directory
                --bindir   : binary directory
                --help     : this help

        FILEiARGV is an array of:
                \<open>    : device open string parameter
                \$<number>  : text formatted as a number
                @<point>   : text formatted as a point
                "<string>  : plain text
                ?<boolean> : text containing string 'true' or 'false'
```

Programul *plint* preia pe rand fiecare fisier sursa *plint* de intrare prin specificarea numelui acestuia, precum si unele argumente care vor fi disponibile apoi prin variabial "argv". Memoria alocat de-a lungul "executiei" unui fisier sursa este complet dealocata la inceperea executiei urmatorului fisier, in aceeasi sesiune (adica fara a iesi din program).

Daca nu se doreste afisarea diferitelor mesaje pe langa outputul programului, se poate directa iesirea standard de eroare spre nul, astfel: "plint 2>/dev/null ...".

Optiunile sunt dependente de modul in care a fost compilata aplicatia; de exemplu, flagul '-d' nu apare mai sus, dar e disponibil cand este validat suportul de debug la compilare; in acest caz '-d' controleaza permisiunea iesirii mesajelor de debug.

Argumentul '-v' poate fi util pentru a afisa mai multe informatii.

Daca se doreste numai verficarea corectitudinii sintactice a codului sursa, se poate comuta oprirea in aceasta faza prin flagul '-p'.

Numele procedurilor si a variabilelor mai des folosite se tin in cache; se poate dezafecta fiecare cache in parte prin comutarea flagurilor '-P', respectiv '-V' [1].

Despre fiecare plugin cu care e compilata aplicatia, se stie numele lui, o scurta descriere si autorul (sau autorii). Daca se doreste obtinerea acestor informatii, se poate folosi optiunea lunga "–plugins". Aceasta va face sa se afiseze numarul de pluginuri compilate, si pentru fiecare plugin in parte, pe line, numele, autorul si descrierea lui.

[1] De fapt, cacheul de variabile nu e implementat inca, deci nu are nici un efect folosirea flagului '-V'; totusi, va fi folosit probabil in versiuni ulterioare

Fiecare tip de plotter suportat de aplicatie are un nume unic; plotterul curent cu care lucreaza aplicatia este afisat prin folosirea optiunii lungi "–plotter"[2].

Momentul terminarii compilarii, tipul masinii si sistemului pe care s-a efectuat compilarea constituie un timbru care se aplica aplicatiei; acesta poat fi vizualizat cu optiunea "–stamp"[3].

Directorul ce gazduieste fisierul binar instalat in sistem, precum si cel care gazduieste bibilioteca de fisiere sursa plint bune de inserat, se poate gasi prin vizualizare din optiunile "–bindir", respectiv "–insdir". Efectul comenzii "plint –bindir" este la fel cu "which plint", daca a fost instalat *plint*.

Fiecarui program sursa plint corect sintactic, daca nu e activ flagul "-p", i se creeaza o variabila numita "argv"[4]; aceasta variabila este un tablou si este read-only si singleton; in ea se pot depune diferite valori chiar din linia de comanda, acestea pot fi numere intregi sau reale, puncte, booleeni, stringuri. Pentru codificarea tipului valorii, se prefixeaza aceasta cu specificatorul corespunzator valorii din cadrul limbajuli *plint*. De exemplu:

```
$ plint test.plt \$0 \"zero \?false \@[0,0]
```

va crea argv echivalent cu literalul tablou !{ 0, 'zero', false, [0,0] }. Deoarece specificatorii constituie metacaractere pentru shell, pentru a nu creea confuzii s-au precedat acestea cu `\`. (s-au trivializat).

Inainte de incepe faza de rulare efectiva, plotterul se initializeaza, si poate primi un argument de tip string. Acesta se poate pasa din linia de comanda, prefixand stringul cu backslash[5].

3.2 Premise

Pentru a intelege codul sursa scris in C mai bine, ar trebui sa se cunoasca anumite tehnici ce s-au folosit in dezvoltarea aplicatiei.

Astfel, toate tipurile au extensia '_t', deoarece in editorul Emacs, aceasta extensie usureaza evidentierea sintaxei.

In C toti identificatorii de variabile/proceduri fac parte din acelasi spatiu de nume (namespace); de aceea, de obicei, un modul al unei aplicatii C are prefixat toate simbolurile exportate (variabile si proceduri nestatice) cu un nume scurt ce identifica modulul, urmat de '_'. La fel aici, "plint_" este un prefix ce apare obsedant.

Deoarece memoria dinamica este des folosita, tipurile declarate sunt mai degraba pointeri la anumite structuri, astfel ca simbolul '*' nu apare prea des in declararea parametrilor sau variabilelor.

plint lucreaza cu valorile variabilelor; iar aceste sunt doar semantic si foarte putin sintactic fundamental diferite; o serie de operatii insa le sunt comune, si acest aspect al similaritatii formei, dar deosebirii fondului este suprins in limbajul C prin folosirea macrourilor ce realizeaza concatenarea argumentelor; de aceea, pentru simplificarea apelurilor acestor macrouri, s-a ales urmatoarea conventie: valorile se denumesc prin 3 litere mici (ex. "str" pentru string); tipurile C, valorile enumerari, numele de variabile, proceduri, numele stivelor, toate trebuie sa aiba, respectiv, totul in comun, in afara de numele tipului valorii. Se va

[2] plotterul fiind de fapt o bibilioteca dinamica, se poate comuta la momentul executiei tipul de plotter folosit

[3] Timbrul e util pentru cunoasterea precisa a contextului aparitiei eventualelor buguri

[4] numele e mostenit din C, perl, specificand tabela de argumente din linia de comanda - arguments' vector

[5] de exemplu, pentru plotterul "X window", poate fi display-ul pe care sa se deschida fereastra, deci poate fi :1

usura enorm de mult modul de prelucrare logic asemanator dar substantial diferit a valorilor; usurare se refera atat la efortul intelectual, cat si la cel fizic de scriere a codului, ce poate fi uneori extenuant si iritant.

Singurul compilator cu care a fost testat aplicati este *gcc*, GNU C compiler. Acesta ofera o serie de "extensii" la standardul limbajului C, printre care:

- operatorul `typeof`, asemanator lui `sizeof`, este un inlocuitor al tipului argumentului - util in macrouri cu concatenari
- initializarea membrilor unei structuri sau uniuni se poate face in orice ordine si nominal, intr-o lista incadrata de acolade, cu fiecare element separat prin virgula de forma: numele campului, ':', valoarea la initializare.
- initializarea a elementelor de la anumiti indecsi dintr-un tablou se poate face sub forma: '[' index ']' '=' valoare; "index" poate fi un element al unei enumerari sau un numar intreg.
- la compilare, in linia de comanda a compilatorului se pot specifica directoarele unde sa se caute fisierele antet. Toate specificarile de fisiere antet din cadrul unitatii de translatare vor fi relative la cel mult una din caile antet cunoscute de compilator; daca nu se gaseste in nici o cale, atunci, deoarece compilatorul schimba directorul curent la cel al fisierului sursa C compilat, se cauta in acest director "curent". De aceea, o modalitate de a imparti logic o aplicatie e de a crea diferite directoare cu nume corespunzatoare functionalitatii cumulate a fisierelor sursa continute. In cadrul fiecarui directoar, toate fisierele sursa din cadrul acestui director vor include fisiere antet tot din cadrul acestui director *fara sa specifice calea directorului relativa la directorul radacina al intregii aplicatii!*. Orice referire la fisiere antet din alte directoare se face sub format:`#include "director/fisier.h"`.

Pentru detalii si alte aspecte consultati "info gcc".

3.3 Ierarhia de directoare

Modul de organizare a ierarhiei de directoare este foarte important pentru usurinta procesului de dezvoltare software.

Asemanator standardului *Unix* de fisiere, radacina *plint* contine urmatoarele directoare:

```
plint/
        bin/
        doc/
        etc/
        lib/
        driver/
        plint/
        plugin/
        plotter/
        share/
        tar/
        tmp/
        tst/
```

plint/

Acesta este directorul ce gazduieste sursele efective ale aplicatiei *plint*. Subdirectoarele sunt:

```
data/
in/
parse/
prog/
misc/
plugin/
```

Directorul *data/* contine sursele legate de "datele" vehiculate de limbajul *plint*, codificate in limbajul C.
Directorul *prog/* contine sursele legate de "datele" vehiculate de interpretorul *plint*, adica cele care suporta rularea (interpretarea) *prog*ramelor.
Directorul *parse/* contine sursele ce ajuta analiza lexicala si gramaticala, adica "parsuirea" codului.
Directorul *misc/* contine diferite (**misc**ellaneous) fisiere utilitare.
Directorul *plugin/* contine sursele adaosurilor care se aduc la aplicatie. Acestea ar trebui sa fie symlinkuri spre directorul real in care sunt implementate. Daca ele sunt acceptate definitiv, se pot transfera in directorul *plint/lib/plugin*.
In subdirectorul *plint/plint/plugin* exista un symlink *lib* → *../../lib/plugin* care poate fi folosit pentru a crea symlinkurile pluginurilor, in genul:

lib/
name.plugin → *lib/name.plugin*
name → *lib/name*

De asemenea, mai exista un symlink *plugin* → *../../plugin* care poate fi folosit pentru a crea symlinkurile pluginurilor imature, in genul:

plugin/
name.plugin → *plugin/name.plugin*
name → *plugin/name*

In fine, directorul *in/* contine sabloanele din care se extind variantele finale ale unor fisiere sursa. Portiunile care se adauga vor fi luate din subdirectoarele directorului *plugin/*, iar versiunile finale vor fi transferata in subdirectoarele *parse/* si *prog/*. Directorul *in/* contine o ierarhie de directoare in "imagine" a directorului parinte.

bin/

Contine unele shell scripturi utilitare, precum si un symlink *plint* → *../plint/plint/plint*.

doc/

Contine aceasta documentatie in directorul *ro/diploma/* scrisa in LaTeX.

etc/

Diverse fisiere auxiliare pot fi plasate aici; unul dintre acestea este un fisier de descriere a sintaxei in vederea evidentierii in editorul folosit (syntax highlighting), anume *mc*.

lib/

O serie de fisiere binare sau text ajung la maturitate, in versiuni finale, bune de folosit; ele sunt stocate in *biblioteci* (**lib**raries). Contine urmatoarele subdirectoare:

```
driver/
filter/
insert/
plotter/
plugin/
primitives/
```

Directorul *driver/* contine sursele diferitelor tipuri de drivere, pentru diferite versiuni ale nucleului Linux, functie de tipul sistemului de fisiere montat in directorul */dev*, pentru dispozitivul proiectat.

Directorul *primitives/* contine fisierul antet de declarare a interfetei de primitive grafice, un "motor" optimizat si bufferizat de transformare a primitivelor in comenzi banale de tipul inainte, inapoi, stai pe loc, creion jos/sus, precum si un test X al acestui "motor".
Subdirectorul *insert/* contine o serie de fisiere sursa *plint*, utile daca sunt inserate sau incluse in alte surse.
Subdirectorul *plugin/* contine acele pluginuri a caror dezvoltare a ajuns la maturitate, si pot fi folosite in construirea unei aplicatii extinse.
Limbajul *plint* fiind grafic, exista o asemanare cu alte limbaje de acest tip, cum ar fi *hpgl* sau *dia*. De aceea sunt utile translatoare intre acestea (sau filtre); directorul *filter/* contine implementarea unor asemenea filtre.
Intre dispozitivele de afisare (plotter, fereastra X) si interpretor exista o interfata concretizata in implementarea unor functii C. Acestea sunt organizate in subdirectorul *plotter/*.

driver/

Driverul curent folosit; este un symlink spre un subdirector din *lib/driver* (de exemplu *driver* $\rightarrow$ *lib/driver/2.4/mknod*)

plotter/

Este un symlink catre "plotterul" curent folosit - *plotter* $\rightarrow$ *lib/plotter/...*

plugin/

Pluginurile care sunt inca in faza de dezvoltare/testare, pot fi stocate aici temporar; totodata este si punctul de pornire (oarecum automatizat) in crearea a unui plugin.

share/

Aplicatia *plint*, modulul curent, plotterul curent si pluginurile folosesc impreuna anumite fisiere; acestea sunt adunate in acest subdirector.

tar/

Versiuni stabile pot fi arhivate si arhivele stocate aici.

tmp/

Diferite fisiere pe care dezvoltatorul le foloseste in diferite procese alaturate celui de dezvoltare, sunt tinute aici.

tst/

Testarea interpretorului se realizeaza de obicei prin rularea unor fisiere sursa *plint* de test; acestea sunt stocate aici. Testarea altor tipuri de programe (de exemplu scrise in limbajul C) ar trebui sa se faca in subdirectorul *tmp/tst/*.

3.4 Miez - Core

Elementele de baza static si dinamice ale limbajului se concretizeaza in structuri de date care sunt prelucrate de niste algoritmi specifici.
Datele vehiculate in limbaj se numesc *variabile*, ele avand in momentul executiei ("runtime") un tip si o valoare. Valorile sunt abordate in sectiunea **Date**.
Structura statica si fluxul dinamic al executiei programului, precum si implementarea tipurilor variabilelor, sunt discutate in sectiunea **Program**.
In fine, partea de analiza lexicala si sintactica a interpretorului este detaliata in sectiunea **Parser**.

3.4.1 Date

Datele unui program sursa *plint* le reprezinta variabilele. O *variabila* are asociata o valoare si un tip. Tipul valoarii este unul dintre urmatoarele posibile (in numar fix):

- nul
- intreg
- real
- punct 2D
- boolean
- string
- tablou de variabile
- colectie de variabile

Dintr-o alta perspectiva, valorile se pot clasifica in domenii si functii, sau scalare si vectoriale; valorile domenii reprezinta, asa cum le spune si numele o multime de valori scalarea (un domeniu de valori); valorile functii pun in corespondenta valori domenii cu alte variabile. Dupa cum s-s spus si la introducere, variabilele modeleaza structuri de date care pot fi indexate, adica exista functii index ce pun in corespondenta valori domenii cu variabile; in acest gen trebuie vazuta sintagma valori domenii - valori functii. Valorile functii sunt ultimele doua amintite in enumerarea de mai sus, iar valorile domenii restul. Nulul este o valoare mai speciala, in sensul ca nu reprezinta nimic util. Facand o analogie cu circuitele electrice, in aplicatia *plint* valoarea nul exista intr-o singura instanta in memorie, adica este un fel de "nul" electric (masa), pe cand restul valorilor (potentialele electricee) putand lua o serie de valori.

Numarul de tipuri de valori se poate extinde (si probabail ca in versiuni viitoare se va intampla acest lucru) prin adaugarea unor proceduri pentru copierea, duplicarea, stergerea, alocarea implicita a unei valori de un anumit tip; se va avea in vedere pe viitor crearea unor valori de tip: data opaca (sir de octeti), fisier text, fisier binar, socketuri, entitati IPC (comunicare intre procese).

Variabilele au asociata o anumita valoare; mai multe variabile pot sa refere aceeasi valoare, si cand i se schimba valoarea la una, aceasta se va reflecta si in cealalta; de asemenea "variabilele" in sensul identificatorilor programului pot sa-si imparta aceeasi valoare; tot asa, valorile functii pot sa nu fie injective.
Tipul unei variabile este de fapt un pointer la o secventa de operatii (interfata) cheie; tipul unei variabile este de fapt o *cheie pentru tip*ul ei; operatiile care trebuiesc implementate pentru ai definii tipul sunt cuprinse intr-o structura de pointeri la functii:

```
 /** @defgroup variable plint variables */
/* @{ */

/**
 * @file plint/data/variable.h
 * @brief variables data structure
 * @author Sebastian Glita
 */

/** the variables' type interface.
 * the type interface is a collection of callback functions called
 * from various key-points; a type structure of a variable must have
 * at least these members at zero offset in the structure
```

Indiferent de valoarea ei, motorul tipului variabilei poate face orice in aceste functii. Tipurile de mai sus se regasesc definite ("typedef"d́) in fisierul*data/types.h*:

```
 /* @addtogroup data */
/* @{ */

/**
 * @file plint/data/types.h
 * @brief plint data typedefs
 * @author Sebastian Glita
 */

typedef struct plint_tag_num plint_num_t;
typedef struct plint_tag_pnt plint_pnt_t;
typedef union plint_tag_2pt plint_2pt_t;
typedef struct plint_tag_str plint_str_t;
typedef int plint_bol_t;
typedef struct plint_tag_tab plint_tab_t;
typedef hash_t plint_hsh_t;

typedef struct plint_tag_idx plint_idx_t;
typedef struct plint_tag_rng plint_range_t;
```

```
typedef union plint_tag_value plint_val_t;
```

Fie ca e valoare sau variabila, o data are alocata o memorie dinamica; cum s-a spus mai sus aceste date pot fi folosite de mai multe alte date deodata. Prin urmare, s-a prevazut un mecanism de *numarare a referintelor* la date ("reference counting"). O data se elibereaza numai cand contorul de referinte ajunge la 0. Duplicarea unei date inseamna incrementarea numarului de referinte. Eliberarea unei date inseamna decrementarea numarului de referinte. Copierea unei date inseamna returnarea unei noi date alocate, care sa fie "copia" *fidela* a unei date initiale; notiunea de *fidel* este interpretabila.

Definirea tipurilor valorilor trebuie sa prevada operatii de conversie la alte tipuri (cu valori implicite daca nu se poate converti), de comparare, de specificare a lungimii (abstract), de incrementare si decrementare (abstract).

Avand in vedere ca printr-o valoare se poate ajunge la alte variabile si ca valorile sunt alocate dinamic, rezulta ca ele pot fi vazut ca memorii pentru variabile.

Urmatoarele trei sectiuni abordeaza cele trei personalitati ale valorilor: domenii, functii, memorii. Mai intai insa, se va vorbi putin despre numararea referintelor.

Referinte

Numararea referintelor este o tehnica care faciliteaza gestiunea curata a memoriei dinamice; astfel, se va sti cu exactitate momentul cand o zona de memorie nu mai este necesara (si va fi deci silentios eliberata (silently discarded)), daca fiecare dorinta de folosire pentru un timp a ei este marcata de incrementarea numarului de referinte, iar cand nu mai e necesara se va decrementa. Totusi, entitatea care foloseste o anumita zona de memorie nu stie de numarul de referinte; stie in schimba de operatia de *duplicare*, respectiv *eliberare*. Duplicarea inseamna ca nu se creeaza o copie noua, iar eliberarea inseamna ca potential poate duce la reala eliberare a memoriei ocupate.

Numarul de referinte a unei zone de memorie se initializeaza la 1 (atentie!) in momentul primei "referiri". Creearea unei zone de memorie este deci urmata implicit de o duplicare.

Zona de memorie a valorii nul este referita prima oara la pornirea executiei programului, fiind singura care nu poate fi "alocata/eliberata" prin programele *plint*.

Avand in vedere ca numarul de referiri ar putea fi destul de mare ca sa depaseasca capacitatea unui cuvant al masinii, trebuie facuta o analiza asupra domeniului de variatie a lui. Daca fiecare cuvant din memorie ar indirecta spre aceeasi data cu numar de referinta, rezulta ca acesta ar putea inca fi reprezentat pe un cuvant (ar numara cate cuvinte sunt in memore), deoarece dimensiunea memoriei este 2 la puterea lungimii cuvantului in biti. Daca insa memoria datei este utilizata in comun de mai multe procese (neimplementat), atunci numarul de referinte are putea depasi capacitatea cuvantului.

Extensia GNU[6] a limbajului C permite definirea unui tip de date *long long* care are capacitate foarte mare. Practic ar fi suficient, deoarece ipoteza de mai sus este un caz teoretic care nu se poate ajunge in practica.

Numarul de referinte mai este folosit si in cadrul algoritmului de colectare a reziduurilor, deoarece acesta foloste propriile numere de referinta pentru tinerea in evidenta a zonelor potenial-reziduuale.

Tipul numarului de referinta este definit in fisierul *share/types.h* in felul urmator:

```
/** 'number of references' type
*/

#if defined(__GNUC__) && defined(NDEBUG)
```

[6] GNU is Not Unix

```
typedef unsigned long long noref_t;
#else
typedef unsigned long noref_t;
#endif
```

Valori domenii

Valorile domenii sunt:

- numere intregi
- numere reale
- puncte 2D
- booleni: adevarat/fals — true/false
- siruri de caractere ASCII

In continuare se detaliaza semnificatia fiecarui tip de valoare, precizandu-se domeniul de valori, observatii legate de implementarea in C, operatiile posbile si conversiile cu alte valori[7].

Numere Numerele pot fi intregi sau reale; domeniul de variatie este corelat cu capacitatea cuvantului masinii (uzual 32 biti) pentru numerele intregi (*int* in sens C); numerele reale sunt de fapt valori *double* in sens C, deci numere in virgula flotanta pe 64 biti. Ambele tipuri sunt cu semn. Tipul propriu zis este:

```
/**
 * @file plint/data/val/number.h
 * @brief number value description
 * @author Sebastian Glita
 */

/** number value.
 * @typedef plint_num_t
 * numbers can be integer (PLINT_num_int) or real (PLINT_dbl)
 * @ingroup data value
 */

struct plint_tag_num
{

  plint_typ_t typ;
```

Operatii [8]

```
#define plint_num_dbl_cpy(__d__) (__d__)
#define plint_num_dbl_dup plint_num_dbl_cpy
#define plint_num_dbl_free(__d__)
```

[7]Aceste chestiuni legate de valori sunt suportate de miezul aplicatiei; extensiile insa pot da propria semnificatie sau extinde operatile

[8]Rezultatul intr-o operatie este intreg daca toti operanzii sunt intregi; altfel e real

```
/*
 * basic operations
 */

int plint_num_nat(plint_num_t);
plint_num_t plint_num_rnd(plint_num_t);
plint_num_t plint_num_min(plint_num_t, plint_num_t);
plint_num_t plint_num_max(plint_num_t, plint_num_t);


/*
 * arithmetic operations
 */

void plint_num_add(plint_num_t *, plint_num_t);
void plint_num_sub(plint_num_t *, plint_num_t);
void plint_num_mul(plint_num_t *, plint_num_t);
void plint_num_div(plint_num_t *, plint_num_t);
void plint_num_rem(plint_num_t *, plint_num_t);
void plint_num_umin(plint_num_t *);
void plint_num_upls(plint_num_t *);
void plint_num_udiv(plint_num_t *);



/*
 * logical operations
 */

void plint_num_and(plint_num_t *, plint_num_t);
void plint_num_or(plint_num_t *, plint_num_t);
void plint_num_xor(plint_num_t *, plint_num_t);
void plint_num_unot(plint_num_t *);



/*
 * comparison operations
 */

int plint_num_equ(plint_num_t, plint_num_t);
int plint_num_neq(plint_num_t, plint_num_t);
int plint_num_lt(plint_num_t, plint_num_t);
int plint_num_gt(plint_num_t, plint_num_t);
int plint_num_lte(plint_num_t, plint_num_t);
int plint_num_gte(plint_num_t, plint_num_t);
```

Conversii

- din/in numar: identic

← din punct: valoarea coordonatei x

→ in punct: x, y iau valoarea numarului

← din boolean: intreg, 1 daca e adevarat, 0 altfel

→ in boolean: adevarat daca e diferit de 0 (respectiv 0.), fals altfel

← din string: lungimea sirului

→ in string: reprezentarea textuala

← din tablou: lungimea tabloului

→ in tablou: implicit

← din hash: implicit

→ in hash: implicit

– implicit: valoarea 0, ca intreg

Puncte Punctele 2D sunt o pereche de numere intregi cu semn, nu in sensul de mai sus; ele reprezinta respectiv coordonatele pe abscisa si ordonata, x si y. Structura este:

```
 /**
 * @file plint/data/val/point.h
 * @brief point value description
 * @author Sebastian Glita
 */

/** point value.
 * @typedef plint_pnt_t
 * 2D points are a pair of integers, x and y coordinates.
 * @ingroup data value
 */



/*
 * "arithmetic" operations
 */

inline void plint_pnt_add(plint_pnt_t *, plint_pnt_t);
inline void plint_pnt_sub(plint_pnt_t *, plint_pnt_t);
void plint_pnt_mul(plint_pnt_t *, plint_pnt_t);
void plint_pnt_div(plint_pnt_t *, plint_pnt_t) __attribute__((weak));
inline void plint_pnt_umin(plint_pnt_t *);
inline void plint_pnt_ucnj(plint_pnt_t *);

/*
 * comparison operators
 */

inline int plint_pnt_equ(plint_pnt_t, plint_pnt_t);
inline int plint_pnt_neq(plint_pnt_t, plint_pnt_t);
inline int plint_pnt_lte(plint_pnt_t, plint_pnt_t);
inline int plint_pnt_gte(plint_pnt_t, plint_pnt_t);
inline int plint_pnt_lt(plint_pnt_t, plint_pnt_t);
```

```
inline int plint_pnt_gt(plint_pnt_t, plint_pnt_t);

/*
 * mixt arithmetic with numbers
 */

void plint_pnt_num_add(plint_pnt_t *, plint_num_t);
void plint_pnt_num_sub(plint_pnt_t *, plint_num_t);
void plint_pnt_num_mul(plint_pnt_t *, plint_num_t);
void plint_pnt_num_div(plint_pnt_t *, plint_num_t);
void plint_pnt_num_rem(plint_pnt_t *, plint_num_t);
```

Conversii

– in/din punct: identic

← din numar: un punct de coordonate egale cu valoarea intreaga a numarului

→ in numar: valoarea coordonatei x

← din boolean: [1,0] daca e adevarat, [0,1] altfel

→ in boolean: adevarat daca nu e origine

← din string: un punct de pe prima bisectoare, la o distanta egala cu lungimea sirului de [0,0]

→ in string: reprezentarea in baza 10 a coordonatelor, separate prin virgula, intre paranteze drepte

← din tablou: un punct de pe prima bisectoare, la o distanta egala cu lungimea tabloului de [0,0]

→ in tablou: un tablou cu 2 elemente numere intregi, corespunzatoare coordonatelor

← din hash: implicit

→ in hash: un hash cu 2 elemente cu cheile "'x"', respectiv "'y"' avand valorile numere intregi corespunzatoare coordonatelor

– implicit: originea [0,0]

Stringuri Stringurile sunt siruri de caractere dintr-un subset al codului ASCII, anume litere, cifre si simboluri (vezi 6.1). Lungimea lor poate fi oricat de mare. Tipul C folosit este:

```
/**
 * @file plint/data/val/string.h
 * @brief string value description
 * @author Sebastian Glita
 */

/** string value.
 * @typedef plint_str_t
 * string is a sequence of ASCII characters (8 bits), C char type, of
     certain length.
```

```
 * @ingroup data value
 */

struct plint_tag_str
{

  size_t n;
  char *s;

#define plint_str_n(__s__) (__s__).n

 #define PLINT_str_INIT_NUL PLINT_str_INIT(0,0)
#define PLINT_str_NEW(__s__, __n__) ((__s__).s = malloc((((__s__).n = (
    __n__)) + 1) * sizeof(char)))
#define PLINT_str_GROW(__s__, __x__) ((__s__).s = realloc((__s__).s, (((
    __s__).n __x__) + 1) * sizeof(char)))
#define PLINT_str_EMPTY(__s__) 0[PLINT_str_NEW(__s__, 0)] = '\0'

};

__BEGIN_DECLS

/*
 * operators & substring
 */

#define plint_str_add plint_str_cat
void plint_str_sub(plint_str_t *, plint_str_t);
void plint_str_rem(plint_str_t *, plint_str_t);
void plint_str_mul(plint_str_t *, size_t);
void plint_str_sed(plint_str_t *, plint_str_t, plint_str_t);
plint_tab_t plint_str_rex(plint_str_t, plint_str_t);
void plint_str_loc(plint_str_t *);
void plint_str_upc(plint_str_t *);

/*
 * comparison operators
 */

int plint_str_inc(plint_str_t, plint_str_t, int);
```

Conversii

- $-$ in/din string: identic
- $\leftarrow$ din numar: reprezentarea in baza 10 a numarului
- $\rightarrow$ in numar: lungimea sirului
- $\leftarrow$ din punct: reprezentarea in baza 10 a coordonatelor, separate prin virgula, intre paranteze drepte
- $\rightarrow$ in punct: un punct de pe prima bisectoare, la o distanta egala cu lungimea sirului de [0,0]

← din boolean: “true” daca e adevarat, “false” altfel

→ in boolean: adevarat daca sirul nu e nul

← din tablou: toate elementele tabloului se scriu in sir, separate prin virgula, intre paranteze drepte

→ in tablou: un tablou de dimensiune egala cu lungimea sirului, avand elemente siruri de lungime 1, constituite din caracterele, pe rand, ale sirului argument

← din hash: toate elementele hashului se scriu in sir, ca perechi separate prin “-¿”, intre paranteze drepte, separate prin virgula, totul intre acolade

→ in hash: un hash cu 1 element cu cheia sirul initial si valoarea tot asa

– implicit: sirul nul “”

Booleeni O valoare booleeana poate fi adevarata sau falsa, mai exact “true” sau “false”. Tipul boolean este “int”.

Operatii Operatiile sunt “inline”, deoarece tipul este intreg, fundamental.

Conversii

– in/din boolean: identic

← din numar: adevarat daca e zero, altfel fals

→ in numar: intreg, 1 daca e adevarat, 0 altfel

← din punct: adevarat daca nu e origine

→ in punct: [1,0] daca e adevarat, [0,1] altfel

← din string: adevarat daca sirul nu e nul

→ in string: “true” daca e adevarat, “false” altfel

← din tablou: adevarat daca tabloul nu e gol

→ in tablou: implicit

← din hash: adevarat daca hashul nu e gol

→ in hash: implicit

– implicit: “false”

Valori functii

- tablouri - arrays
- colectii - hashes

Tablouri Un tablou pune in corespondenta variabile unor indici numere intregi incepand cu 1 si pana la “dimensiunea” tabloului. Declararea tipului este:

```
/**
 * @file plint/data/val/table.h
 * @brief array value description
 * @author Sebastian Glita
 */

/** array value.
 * @typedef plint_tab_t
 * an array is a memory which maps consecutively integer indexes to
     variables.
 * a table can be infinitely indexed; it's not "cubic" like N1xN2xN3x...
     Nn
 * @ingroup data value
 */

struct plint_tag_tab
{

  size_t n;    /* dimension */
  plint_var_t *a; /* array pointer */

#define PLINT_tab_INIT(__n__) { a:malloc((__n__) * sizeof(plint_var_t
    )), n:(__n__) }
#define PLINT_tab_INIT_NUL { a:0, n: 0 }
#define PLINT_tab_EMPTY(__t__) ((__t__).a = 0, (__t__).n = 0)
#define PLINT_tab_NEW(__t__, __n__) ((__t__).a = malloc(((__t__).n = (
    __n__)) * sizeof(plint_var_t)))
  /* table grow is only allowed if the table IS NOT A MEMORY (i.e., if
      it's not part
```

Operatii Fiind un container de variabile, se poate aplica asupra elementelor sale operatii, deci mai are in plus o serie de operatii de acest gen.

```
       the discussion is general for reallocation issue, because tables
           ARE MEMORIES.
   */

/*
 * arithmetic operators
 */

#define plint_tab_add plint_tab_cat
void plint_tab_mul(plint_tab_t *, size_t);

/*
 * comparison operators
 */

int plint_tab_equ(plint_tab_t, plint_tab_t);
int plint_tab_neq(plint_tab_t, plint_tab_t);


/*
 * has and includes operator
```

```
 */

int plint_tab_lt(plint_ptr_t *, plint_tab_t *);
int plint_tab_gt(plint_tab_t *, plint_ptr_t *);
int plint_tab_lte(plint_ptr_t *, plint_tab_t *);
int plint_tab_gte(plint_tab_t *, plint_ptr_t *);

/*
 * applying a function for each element with an external value
 */

void plint_tab_num_add(plint_tab_t *, plint_num_t);
void plint_tab_num_sub(plint_tab_t *, plint_num_t, int);
void plint_tab_num_mul(plint_tab_t *, plint_num_t);
void plint_tab_num_div(plint_tab_t *, plint_num_t, int);
void plint_tab_num_rem(plint_tab_t *, plint_num_t, int);
void plint_tab_num_and(plint_tab_t *, plint_num_t);
void plint_tab_num_or(plint_tab_t *, plint_num_t);
void plint_tab_num_xor(plint_tab_t *, plint_num_t);

void plint_tab_pnt_add(plint_tab_t *, plint_pnt_t);
void plint_tab_pnt_sub(plint_tab_t *, plint_pnt_t, int);
void plint_tab_pnt_mul(plint_tab_t *, plint_pnt_t);
void plint_tab_pnt_div(plint_tab_t *, plint_pnt_t, int);

void plint_tab_str_add(plint_tab_t *, plint_str_t, int);
void plint_tab_str_sub(plint_tab_t *, plint_str_t, int);
void plint_tab_str_rem(plint_tab_t *, plint_str_t, int);

void plint_tab_bol_and(plint_tab_t *, plint_bol_t);
void plint_tab_bol_or(plint_tab_t *, plint_bol_t);
void plint_tab_bol_xor(plint_tab_t *, plint_bol_t);
```

Conversii

- – in/din tablou: identic
- ← din numar: implicit
- → in numar: lungimea tabloului
- ← din punct: un tablou cu 2 elemente numere intregi, corespunzatoare coordonatelor
- → in punct: un punct de pe prima bisectoare, la o distanta egala cu lungimea tabloului de [0,0]
- ← din boolean: implicit
- → in boolean: adevarat daca tabloul nu e gol
- ← din string: un tablou de dimensiune egala cu lungimea sirului, avand elemente siruri de lungime 1, constituite din caracterele, pe rand, ale sirului argument
- → in string: toate elementele tabloului se scriu in sir, separate prin virgula, intre paranteze drepte

← din hash: un tablou de lungime egala cu dimensiunea hashului, avand ca elemente, intr-o *ordine oarecare*, un tablou cu 2 elemente, cheia si respectiv valoarea din hash

→ in hash: un hash de dimensiune egala cu lungimea tabloului, avand ca elemente perechi cheie-valoare cu valoarea egala cu cea de la un indice in tablou, iar cheia egala cu indicele sub forma de string

– implicit: un tablou cu un singur element, valoarea argument

Hashuri Acest tip de hash este folosit si pentru stocarea numelor variabilelor. Ca valoare insa, el are o particularitate, anume pentru fiecare string o valoare hash are o singura data asociata, anume o variabila.

*/

__BEGIN_DECLS

```
plint_hsh_t plint_hsh_new();

#define plint_hsh_add plint_hsh_cat
void plint_hsh_sub(plint_hsh_t, plint_hsh_t);
void plint_hsh_mul(plint_hsh_t, plint_hsh_t);
#define plint_hsh_div plint_hsh_sub
#define plint_hsh_or  plint_hsh_add
#define plint_hsh_and plint_hsh_mul

/*
 * comparison operators
 */

inline int plint_hsh_equ(plint_hsh_t, plint_hsh_t);
inline int plint_hsh_neq(plint_hsh_t, plint_hsh_t);


/*
 * has and includes operator
 */

inline int plint_hsh_lt(plint_hsh_t, plint_hsh_t);
inline int plint_hsh_gt(plint_hsh_t, plint_hsh_t);
inline int plint_hsh_lte(plint_hsh_t, plint_hsh_t);
inline int plint_hsh_gte(plint_hsh_t, plint_hsh_t);
```

Conversii

– in/din hash: identic

← din numar: implicit

→ in numar: implicit

← din punct: un hash cu 2 elemente cu cheile "'x"', respectiv "'y"' avand valorile numere intregi corespunzatoare coordonatelor

→ in punct: implicit

← din boolean: implicit

→ in boolean: adevarat daca hashul nu e gol

← din string: un hash cu 1 element cu cheia sirul initial si valoarea tot asa

→ in string: toate elementele hashului se scriu in sir, ca perechi separate prin "-¿", intre paranteze drepte, separate prin virgula, totul intre acolade

← din tablou: un hash de dimensiune egala cu lungimea tabloului, avand ca elemente perechi cheie-valoare cu valoarea egala cu cea de la un indice in tablou, iar cheia egala cu indicele sub forma de string

→ in tablou: un tablou de lungime egala cu dimensiunea hashului, avand ca elemente, intr-o *ordine oarecare*, un tablou cu 2 elemente, cheia si respectiv valoarea din hash

– implicit: un hash fara nici o pereche cheie → *variabila*

Valori memorii

Memoriile sunt valori care aloca spatiu dinamic pentru a referi variabile. Exista 2 tipuri de valori:

- tablouri
- colectii

Memoriile se gestioneaza special deoarece si ele pot fi valori ale variabilelor si deoarce, pentru a schimba valorile variabilelor e nevoie de modificarea pointerilor spre variabile, deci e nevoie de *referinte* la acesti pointeri, altfel spus de *referinte la variabile*. Daca o astfel de referinta pointeaza intr-o memorie in sensul de mai sus, si memoria e valoare a unei variabile, si daca variabila respectiva ajunge sa aiba numarul de referinte zero, deci se elibereaza, si daca valoarea ei (memoria) ajunge cu numarul de referinte zero si se elibereaza, referinta de la inceput va deveni invalida, fara ca acest lucru sa se observe. E cazul clasica al efectelor laterale in atribuiri:

```
x[1] = { x = 3; }
```

In acest exemplu, prima atribuire se amana pe mai tarziu, pana dupa ce valoarea returnata de blocul din dreapta va fi fost calculata. Se salveaza deci referinta la x[1], sa zicem x[1], se intra in blocul din dreapta si se elibereaza memoria valoarea a lui x, in care x[1] este o referinta, pentru a i se atribuie lui x o valoare scalara, devenind intregul 3. Pointerul x[1] va fi invalid.

De aceea, solutia este ca inainte de a intra in blocul din dreapta, sa se salveze memoriile (incrementeze numarul de referinte a valorilor) tuturor referintelor din lista din stanga simbolului de atribuire. La iesirea din bloc, se va efectua o asignare la referinta, dupa care se va elibera memoria; aceasta insa va avea numarul de referinte zero, fiind deci silentios eliberata.

Ca si consecinta, efectele laterale sunt bine suportate in *plint*, efectul lor negativ fiind doar atribuiri fara sens.

Asignari

Exista doua tipuri de asignari: prin valoare si prin referinta. Asignarea prin referinta e sub forma: ‘variable’ = ‘expression’; Asignarea prin valoare arata astfel: =‘variable’ = ‘expression’;

Structurile de date ale variabilelor arata asa[9]:

```
  Var1                    Var2
+------+                +------+
|   *---.                |  *   |
+------+ \                +--|---+
          \               /
           \   +-----+   /
1           `->|  2  |<-'
               +-----+
               |  *  |
               +--|--+
                  |
                  |
                  v
               +-----+
2              |  2  |
               +-----+
               | INT |
               +-----+
               | 100 |
               +-----+
                  ^
                  |
               +--|--+
               |  *  |
               +-----+
               |  1  |
               +--|--+
                  ^
  Var3            |
+------+          |
|   *-------------'
+------+

               +-----+
3              |  1  |
               +-----+
               | PNT |
               +-----+
               |[0,0]|
               +-----+
                  ^
                  |
               +--|--+
               |  *  |
               +-----+
4              |  1  |
```

[9]ignorati numerele din stanga, deocamdata

```
               +--|--+
                  ^
  Var4            |
+------+          |
|   *-------------'
+------+

Var1 == Var2 == Var3 == 100
Var4 == [0,0]
```

Daca facem ‘Var2 = Var4’, 2-ul din linia 1 se va schimba in 1, iar 1-ul din linia 4 se va incrementa. Daca mai departe facem ‘Var3 = Var5’, the 1-ul din linia 1 se va schimba in 0, decrementandu-se astfel 2-ul din linia 2. Iata ce se obtine:

```
Var1 == 100
Var2 == Var4 == [0, 0]

               +-----+
3              |  1  |
               +-----+
               | PNT |
               +-----+
               |[0,0]|
               +-----+
                  ^
                  |
               +--|--+
               |  *  |
               +-----+
4              |  2  |<----------.
               +--|--+           |
                  ^              |
  Var4            |              |
+------+          |              |
|   *-------------'              |
+------+                         |
                                 |
  Var1                    Var2   |
+------S                +------S |
|   *---.                |  *-----'
+------S \                +------S
          \
           \   +-----+
1           `->|  1  |
               +-----+
               |  *  |
               +--|--+
                  |
                  |
                  v
               +-----+
2              |  1  |
               +-----+
               | INT |
               +-----+
               | 100 |
               +-----+
```

Totusi, daca am fi incecput prin '=Var2 = Var4', 2-ul din linia 1 nu se va schimba, 2-ul din linia 2 se va decrementa si 1-ul din linia 3 se va incrementa:

```
Var2 == Var1 == Var4 == [0,0]
Var3 == 100

  Var1                    Var2
+------S                +------+
|   *---.                |  *   |
+------S \                +--|---+
          \               /
           \   +-----+   /
1           `->|  2  |<-'
               +-----+
               |  *  |
               +--|--+
                  |
                  `--------.
                           |
               +-----+     |
2              |  1  |     |
               +-----+     |
               | INT |     |
               +-----+     |
               | 100 |     |
               +-----+     |
                  ^        |
                  |        |
               +--|--+     |
               |  *  |     |
               +-----+     |
               |  1  |     |
               +--|--+     |
                  ^        |
  Var3            |        |
+------S          |        |
|   *-------------'        |
+------S                   |
                           |
               +-----+     |
3              |  2  |     |
               +-----+     |
               | PNT |<----'
               +-----+
               |[0,0]|
               +-----+
                  ^
                  |
               +--|--+
```

```
               |  *  |
               +-----+
4              |  1  |
               +--|--+
                  ^
  Var4            |
+------S          |
|   *-------------'
+------S
```

Colectarea reziduurilor

Toata memoria dinamica in timpul executiei unui program *plint* este gestionata prin numere de referinte. Pornind de la variabilele *statice* numite *contextul programului*, se poate tine evidenta fiecarui octet de memorie alocat, prin corecta gestionare a zonelor contiguue si inlantuite de memorie. In teoria "colectoarelor de reziduurui" variabilele statice constituie setul de baza (*root set*) de la care pornind, se poate ajunge mai apoi la orice octet dinamic alocat, bine gestionand inlantuirile.

Exista insa un caz capcana care genereaza zone inlantuite in memorie (lanturi) la care nu se mai poate ajunge pornind de la setul de baza, deci nu mai pot fi dezalocate, ducand la pierde de memorie. *plint* interpretand succesiv toate fisierele sursa din linia de comanda in cadrul aceluiasi proces, aceasta poata duce la epuizarea foarte rapida a memoriei.

Cauza care sta la baza aparitiei acestor reziduuri este legata de numarul de referinte ale variabilelor si valorilor functii; deoarece o variabila cu valoare functie poate indirecta printr-un indice spre o *alta* variabila, nu este obligatoriu ca cele doua variabile sa fie diferite, adica se poate crea o inlantuire in cerc inchis, sau se mai poate spune *lant de referinte*.

Hash

Hashul este o structura de date care pastreaza o singura data un sir de caractere ASCII si ofera ulterior posibilitatea unei cautari rapide a sirului. Un hash poate gestiona mai multe spatii de nume (*namespace*) deodata, astfel incat daca se cauta un string, trebuie precizata si cheia catre spatiul de nume dorit. Odata cu un sir, in hash se introduce si o data proprie acestui sir in spatiul de nume dorit. Data aceea se poate regasi mai apoi.

Exista posibilitatea iterarii tuturor sirurilor dintr-un spatiu de nume sau din intreg hashul, precum si a iterarii tuturor datelor pentru un acelasi sir, din intreg hashul sau dintr-un namespace particular.

Valoarea hash din programele plint este un exemplu de folosire intr-un mod particular al hashului, adica cu o singura data per string, deci nu se permit mai multe siruri. Este usor de detectat aceasta situatie, determinand daca indexul pentru un anumit string in hash e valid sau invalid.

Hashul este folosit si ca loc de stocare a datelor variabilelor din programele sursa, intr-un spatiu de nume al variabilelor automate (pe stiva); datele asociate lor sunt memorii ce contin o singura referinta la variabila. un alt spatiu de nume este create pentru argumentele formale ale procedurilor.

Pluginurile "static" si "class" adauga la randul lor spatii de nume la hashul programului (deci folosit si de spatiul de nume al variabilelor pe stiva si a argumentelor formale). Primul adauga un spatiu de nume pentru variabilele statice. Pluginul "class" adauga cate doua spatii de nume pentru fiecare clasa nou creata, una pentru numele metodelor membru, alta pentru numele datelor membru.

3.4.2 Parser

Stive

Se cunoaste ca majoritatea arhitecturilor procesoarelor sunt de tip “stack machine”; adica exista o stiva de memorie care e folosita pentru a tine date intr-un proces de avansare in timp in “subrutine imbricate”; aceste date sunt date temporare utile, parametrii de apel si adrese de revenire.

Fie spre exemplu procedura urmatoare, in limbajul C: language=C define wait getchar define a () define minute ; include ¡stdio.h¿ void recursive (int n) auto int i, double d, char c; i = n * n; d = i/(n+1); c = ’0’+n; if (n) recursive (n-1); else wait a minute /* si apelul */ ... recursive (2);

Cand n va ajunge 0, in asteptarea apasarii unei taste, in urma apelurilor recursive, stiva arata cam asa:

STIVA
-asteptare tasta-
-adresa revenire- i=0 d=0. c=’0’ n=0
-adresa revenire- i=1 d=0.5 c=’1’ n=1
-adresa revenire- i=4 d=1.3333333333333 c=’2’ n=2
-adresa revenire-

Daca acum impartime stiva unica in mai multe stive, cate una pentru fiecare slot din stiva initiala, putem obtine ceva de genul:

adr	i	d	c	n
-adresa revenire-	0	0.	’0’	0
-adresa revenire-	1	0.5	’1’	1
-adresa revenire-	4	1.33	’2’	2

Aceasta este metoda folosita in *plint*. Generatorul *bison* fiind un automat finit nerecursiv iterativ si fiind scris in C, nu ofera suport pentru dealocarea automata a stivei proprii de neterminale si terminale, de exemplu la aparitia unui semnal. Acest deficit este surmontat de folosirea unui modul care se ocupa numai de stivele variabilelor ce intr-o varianta recursiva ar fi fost automate.

Astfel exista cate o stiva pentru variabile, referinte, valori, stringuri, tabele, hashuri, indecsi etc.

Aceste stive, in caz ca apare ceva neprevazut care intrerupe brusc analiza gramaticala (deci interpretarea), vor fi eliberate in final, eliberandu-se memoria pe care o folosesc elementele lor.

3.4.3 Program

Din puncte de vedere static, un program este un arbore de blocuri; acestea provin din fie direct din fisierul sursa, fie din alte fisiere incluse. Structura este deci colectia de reguli privitoare la blocuri si fisiere.

Din punct de vedere dinamic, fluxul unui program este o stiva de instante ale blocurilor-nod din arborele static. Un bloc-nod poate avea mai multe instante pe stiva, cea mai de sus fiind ultima apelata.
Stiva prevede prin definitie posibilitatea existentei unei liste de variabile, pentru fiecare instanta a unui bloc;

Cele doua structuri de date pot identifica, pana la un punct, cele doua faze ale executiei unui program sursa *plint*: verificare si constructie sintactica, respectiv executie semantica. In prima faza, se prelucreaza numai*nu chiar asa* arborele de blocuri, in a doua numai stiva de instante.

Tipurile de date ale programului sunt urmatoarele:

```
 typedef struct plint_tag_list *plint_list_t;
typedef struct plint_tag_proc plint_proc_t;
typedef struct plint_tag_block plint_block_t;
typedef struct plint_tag_tree *plint_tree_t;
typedef struct plint_tag_insprnt *plint_insprnt_t;
typedef struct plint_tag_insert *plint_insert_t;
typedef struct plint_tag_include *plint_include_t;
typedef struct plint_tag_include_stack *plint_include_stack_t;
typedef struct plint_tag_stack *plint_stack_t;
```

Structurile de date detaliate sunt:

lista de variabile Este o lista simplu inlantuita de variabile (memorii la variabile) pe stiva (home); numele variabilelor este index in tabela de hash a programului, iar data propriu-zisa este o memorie tablou cu o variabila. Mai exista un flag care spune daca variabila a fost folosita sau nu, sub forma de avertisment.

```
struct plint_tag_list
{

  plint_stack_t home;
  plint_list_t next;
  hashindex_t name;
  plint_mem_t var;

  int touch:1;
```

semnatura procedura Specifica numele si aritatea (numarul de argumente); numele face parte din namespaceul global din hashul programului:

```
  };

struct plint_tag_proc
{
  hashindex_t name /* if bad this isn't a procedure */, *args /* if
       0 no arguments */;
  size_t arrity;

  /* even if name is bad, comparison is valid */
#define PROC_SIG_EQU(__x__, __y__) (HASH_INDEX_EQU((__x__).name, (
    __y__).name) && (__x__).arrity == (__y__).arrity)
```

nod arbore Despre un nod arbore (tree) se stie nivelul sau, nodul parinte, nodurile fii, daca e procedura, semnatura, tipul blocului, tipul secventei de apel (metoda, normala sau procedura), un flag folosit pentru a cunoaste pe parcursul parsarii si executiei ce informatii sunt disponibile (valide) din aceasta structura; pe parcursul executiei pastreaza si o referinta la instanta curenta de pe stiva:

```
struct plint_tag_tree
{
  int level;

  plint_tree_t prnt, *chlds /* 0 if leaf */, host /* hosting
     procedure */;
  int nchlds;

  plint_proc_t proc;

  char *srcfile /* where code is held */;

  plint_eblk_t blk;
  plint_frame_t frame;

  int flag;  /* -1(11111111111111111111111111111111) : insertion
     point, still parsing
        * -2(11111111111111111111111111111110) : insertion point,
           parsing done
        * +1(00000000000000000000000000000001) : normal point    ,
           still parsing
        * +2(00000000000000000000000000000010) : normal point    ,
           parsing done
        * +4(00000000000000000000000000000100) : normal point    ,
           parsing done, was called
        * +6(00000000000000000000000000000110) : normal point    ,
           parsing done, was called and its most recent instance
           written on the stack backtrace
        */

#define PLINT_TREE_NOPRC(__r__) HASH_INDEX_DOBAD((__r__)->proc.name
   )
#define PLINT_TREE_ISPRC(__r__) (!HASH_INDEX_ISBAD((__r__)->proc.
   name))
#define PLINT_TREE_ISINS(__r__) ((__r__)->flag & ~(~(unsigned int
   )0 >> 1))
#define PLINT_TREE_ISPRS(__r__) ((__r__)->flag & ~(~(unsigned int
   )0 << 1))
#define PLINT_TREE_ISWRP(__r__) ((__r__)->blk == PLINT_BLK_WRAP)

  /* what is ~(~0 << 1) ? :-) */

  FILE *file;

  plint_stack_t top; /* which is the topmost instance of this block
      on the stack */

  /* block type specific, but shared once between instances */
  void *data;
```

```
#define PLINT_TREE_DATA(__r__) (__r__)->data
```

nod stiva lstinputlisting[first=186,last=200]plint/prog/data.h Un nod de pe stiva este o instanta a unui nod arbore. Membrul “block” contine un pointer “.src” de tip “plint_-tree_t” catre nodul arbore corespunzator. Toate variabilele de pe aceasta instanta a unui bloc se tin intr-o lista ce incepe cu “head”, care este variabila implicita a oricarui bloc; in caz ca e metoda, blocul mai contine o variabila speciala, anume “self”. In afara de aceste cazuri, variabilele normale se tine mai incolo in lista, fiind in numar de “count”. Stiva e dublu inlantuita pentru a permite vizitarea usoara a nodurilor ei. Membrul “proc” reprezinta instanta cea mai recenta de pe stiva a procedurii celei mai apropiate care inglobeaza blocul al carui instanta este nodul stiva din varf.

context program De aici se pleaca in vizitarea/prelucrarea/modificarea structurilor de date ce tin evidenta arborilor si stivelor:

```
This is how it looks: a source file like:

 plot name {
     ...
 }

has ... inside a (top) block which is inside the (top) procedure
    'name'.
```

Statica

```
// file a
procedure a () { echo 'Hi, I\'m a'; }

// file b
{
  insert './a'
  a ();
}

// file prg.plt
plot 'prg' {
  insert './a'
  insert './b'
}
```

Arborele arata astfel:

```
+-----------+-----+
| procedure | prg |
+-----------+-----+
        |
        |
        v
        +-----------+---+
        | procedure | a |
        +-----------+---+
        |
```

```
          |
          v
          +----------------+
          | block | noname |
          +----------------+
                  |
                  |
                  v
                  +-----------+---+
                  | procedure | a |
                  +-----------+---+
```

dar de fapt e implementat cu reutilizare:

```
+-----------+-----+
| procedure | prg |
+-----------+-----+
          |
          |
          v
          +-----------+---+
          | procedure | a |
          +-----------+---+<-+
          |                  |
          |                  |
          v                  |
          +----------------+ |
          | block | noname | |
          +----------------+ |
                  |          |
                  |          |
                  +----------+
```

Cazul inserarilor recursive de genul

```
insert './b'

// file b
insert './c'

// file c
insert './a'
```

este detectat cu succes ca si o eroare.

Arborele are urmatoarea structura si/sau urmatorul comportament:

- un nod este un bloc anonim, nominabil, infasurator sau punct de insertie
- fiecare nod are *un* singur pointer catre nodul parinte
- orice nod poate fi un nod parinte, cu un numar oarecare de copii
- drumul catre procedura principala (prin pointerul "prnt") plecand de la un nod interior se schimba in timpul executiei daca intalneste cel putin un punct de insertie, altfel e constant.

- la un moment dat in timpul executiei, exista mai multe drumuri de la un punct de insertie catre procedura principala, insa cel mult unul este activ; restul sunt salvate pe o stiva, urmand sa fie restaurate in ordine inversa
- copii reali ai unui nod care nu e punct de insertie sunt cei directi care nu sunt puncte de insertie, precum si nodurile descendente la care se poate ajunge numai prin puncte de insertie

Dinamica

Ceea ce este interesant la nodurile arborelui structura al programului e ca aceste noduri, blocurile de fapt, implementeaza ele insele anumite proceduri callback (feedback handlere) care sunt apelate in circumstante speciale in timpul executiei, anume atunic cand se intra in bloc, se iese in bloc sau se abandoneaza un bloc. Tipul blocului identifica aceste proceduri. Dupa cum aceastea considera, se ia decizia corespunzatoare; se poate deci modifica comportamentul dinamic al programului prin crearea unor tipuri de blocuri noi si implementarea handlerelor respective. Asa functioneazaa pluginurile gen: “for”, “while”, “loop”, “foreach”.

Fiecare nod arbore are un anumit tip; tipul respectiv identifica o interfata de proceduri callback, care se apeleaza in momentele cheie ale existentei nodului arbore; fiecare nod arbore poate stoca un pointer generica care va putea fi folosit in aceste functii callback pentru comunicare usoara. Momentele cheie ale unui nod arbore sunt:

faza de parse momentul imediat urmator crearii, momentul imediat precedent eliberarii, momentul atasarii la parinte, momentul detasarii de parinte, momentul cand i se ataseaza un “copil”, momentul cand i se detaseaza “un copil”

faza de run momentul cand se adauga o variabila intr-o instanta a acestui bloc si cand se cauta o asemenea variabila; aceest ultim caz se intampla dupa cautarea standard:

```
   p = plint_ref_self(0, mem);
else if (plint_str_equ(str, plint_argv_str))
  p = plint_ref_argv(mem);
else if (plint_str_equ(str, plint_argo_str))
  p = plint_ref_argo(mem);
else
  {
    hashindex_t hi = hash_getindex(plint->prog.hash.id,
        plint_str_sn(str));

    p = (HASH_INDEX_ISBAD(hi)) ? (plint_ref_t)0
:({
```

Cache

Cacheul este o memorie asociativa de regasire rapida a informatiei identificata in mod unic de o eticheta. Cacheul folosit in *plint* este un cache cu algoritm LRU de determinare a intrarii din cache ce se elibereaza pentru a fi alocata unei informatii “mai noi”.

Concret, pentru fiecare intrare valida se cunoaste un numar care e cel mult egal cu numarul maxim de intrari valide si e unic pentru fiecare intrare. Cu cat e mai mare acest numar, cu atat e mai putin probabil sa fie LRU (least recently used — care a fost folosit cel mai demult). De fiecare data cand se gaseste cu success in cache o intrare valida, se decrementeaza contoarele care sunt mai mari decat acesta, care se seteaza la maxim. Daca nu a fost 0, candidatul LRU ramane acelasi; altfel apare un nou LRU, deoarece cel anterior devine brusc MRU (most recently used).

Plint dispune de 2 cacheuri : variabile si proceduri. Cel de variabile nu e inca implementat.

Cacheul de proceduri tine informatii despre corespondenta dintre numele procedurilor care sunt vizibile din blocul curent si structura corespunzatoare de date; la comutarea dintr-un bloc in altul, exista mai multe moduri de invalidare a intrarilor din cache, astfel incat sa reflecte corect noua situatie.

3.4.4 Tip

Tipul variabilelor este gestionat in fiserele prog/type.c si prog/type.h. Aici se implementeaza o subclasa in sens C (structura ce inglobeaza alta structura la offsetul 0), numita “plint_type_info_t”, a cheii unei variabile, care adauga o serie de informatii despre tipul variabilei, dintre care cea mai importanta este o interfata de functii numit *hook*. Aceasta este o colectie de functii callback asociate operatorilor posibili din limbaj (new, echo, +=, -=, $ etc.); un hook este practic tipul unei variabile; aceste tipuri se pot crea/adauga dinamic, stocandu-se lista dublu inlantuita; fiecare tip poate sa precizeze in momentul inregistrarii la multimea existenta de tipuri acele tipuri de care el depinde intr-un fel oarecare. Daca nu se pot asigura dependendentele, atunci inregistrarea este fara succes.

Odata inregistrat un tip, se pot crea variabile de acest tip. Cu aceste variabile se pot forma expresii cu operatori; in momentul aplicarii unui operator pe o variabila, cursul programului poate devia prin apeluri de metode *plint* care sa returneze valoarea rezultata in urma aplicarii operatorului. Alta posibilitate este de a apela metode care returneaza numele unei metode care realizeaza in cele din urma operatia dorita. Mai simplificat, se poate ca rezultatul sa fie calculat direct de “motorul” C al tipului.

Oricare ar fi numarul de operanzi ai unui operator, exact unul dintre acestea este cel pe care se aplica operatorul, iar restul sunt un fel de paramterrii.

Pentru a realiza aplicarea unui operator, dupa momentul citirii din fisier a operanzilor si operandului, se apeleaza dependent de operator niste functii de initializare a operatiei. Aici se va determina cum va fi realizata operatia: imediat, prin apel direct sau indirect (cautare/apel); In cazul operatiilor cu mai multi operanzi (binare), avand in vedere ca exista mai multe tipuri inregistrate la un moment dat, care nu depind unele de altele, este normal sa se verifice acum daca are rost sa se efectueze operatia intre operanzi de diferite tipuri. O modalitatea simpla este de a verifica liniar daca in lista de dependente a operatorului principal exista tipuri ale operanzilor parametrii.

Dupa o anumita experienta obtinuta din teste, mai ales din pluginul “class”, s-a ajuns la concluzia ca verificare dependentelor prin cautare liniara a cheii proprii variabilei este un mod static de a privi lucrurile. S-a corectat acest comportament in doua sensuri:

- cheia cu care operatorii participa la testarea dependentelor nu va fi neapart ce proprie, ci cea returnata de un handler de cheie al tipului (care se apeleaz prin cheia proprie totusi).
- determinarea dependentelor intre doi operanzi (doua tipuri) este facuta de asemenea prin apelul unei funcii callback

3.5 Adaos - Plugins

3.5.1 Bltin

Pluginul "bltin" creeaza si inregistreaza tipul "bltin" al variabilelor, fiind necesar, fara el nu se poate folosi nici miezul interpretorului.

Implementarile operatorilor ofera posiblitatea operarii cu identificatori fara a folosi specificatori, lucru care face mai agreabil limbajul.

3.5.2 Static

Acest plugin creeaza o stiva cu un singur nod (static) pentru fiecare bloc nod. Intervine in handlerele de creareare/eliberare bloc, cautare si validare aduagare variabile.

3.5.3 If-Else

Introduce 2 tipuri de blocuri: "if" si "else". La intrarea intr o cascada de "if-else", se reseteaza 2 variabile: valoaarea de adevar a conditiei, respectiv daca s-a atins vreo varianta corecta. Conditia ca sa se atinga o varianta corecta e ca valoarea de adevar sa fie adevarat cand se intra intr-un bloc "if" si falsa cand se intra intr-un block "else".

3.5.4 Operator

Atunci cand se caut semnatura unei metode corespunzatoare unui operator, aceste se transmite ca argument metodei de cautare sub forma unui string. Alegerea reprezentarii nu ar trebuie sa-l priveasca pe programator. De aceea, acest plugin degreveaza programatorul de a cunoaste cum sa reprezinte operatorii, prin aceea ca defineste niste reguli care folosesc operatorii ca reprezentare a lor literala si nu string, insotite de cuvantul cheie operator.

3.5.5 ForEach

Iteratia "foreach" este implementata prin folosirea unei variabile proprii blocului foreach, tablou sau hash, functie de tipul acestuia, precum si a unui indice de iteratie, intreg, respectiva marca in hash.

La fiecare terminare a executiei blocului se trece la urmatorul element, variabila implicita a blocului fiind un tablou cu 2 elemente: indicele curent si elementul de la indicele curent.

3.6 Plottere

3.6.1 PLINT

Plotterul PLINT este cel realizat practic pentru aceasta lucrare de diploma; despre el se vorbeste in detaliu in sectiunea 27.

3.6.2 X window

Plotterul X window s-a folosit pe parcursul dezvoltarii pentru testarea aplicatiei. El afiseaza toate desenele intr-o fereastra X, fiind un fel de simulare al plotterului special.

Capitolul 4

Plotterul PLINT

Un plotter[1] este un dispozitiv de desenare automat ce imita actiunea unui desenator uman: plasarea si ridicarea unui creion pe/de pe hartie, intr-o asemenea ordine incat contrastul dintre urma "neagra" a creionului si fondul "alb" al hartiei creeaza ochiului uman perceptia unei "schite".

Automatizarea poate fi directa (in timp real) sau indirecta. Cea directa presupune corelarea miscarii manuale a unui desenator (sau, de ce nu, automata a unui alt plotter) de cea "in imagine" a unui (alt) plotter; un exemplu de acest gen este taietorul de tabla comandat de un alt dispozitiv asemanator unui plotter prevazut cu un cap optic ce urmareste un desen. Desenarea in acest caz este "taierea" tablei. Plotterele indirecte presupun un limbaj intermediar care permite desenatorului sa isi "scrie" desenul in vederea desenarii automate, si interpretorului sa comande automat plotterul.

In orice caz, un plotter e caracterizat de existenta a doua miscari independente pe X si pe Y, a caror pozitii relative la un punct fix dau coordonata curenta a creionului in planul desenului precum si de existenta a doar 2 pozitii pe axa Z, anume 0 (atinge hartia) sau non-zero (nu atinge hartia). Daca pozitia pe axa Z poate lua mai mult de 2 valori, atunci plotterul devine *inscriptor*.

Avantajul utilizarii plotterelor in locul imprimatelor e ca trasarea contururilor se face vectorial si nu rasterizat, fiind ideal in cazul desenelor geometrice. Tinand cont de cunostintele matematice in privinta aproximarii curbelor de orice forma prin formule matematice, se poate extinde domeniul desenelor si in zona artistica, ramanand in zona "fara tremur" a editarii de text.

Miscarile pe axele X si Y sunt realizate prin intermediul motoarelor comandate electronic de la un sistem de calcul. Miscarea pe axa Z poate fi simplificata prin folosirea unui electromagnet.

Unele plottere pot utiliza mai multe culori in realizarea desenului, utilizand, de exemplu, creioane de diferite culori, sau stilouri ale caror rezervoare de cerneala se poate alege la un moment dat intre mai multe diferite culori. Si una si alta dintre variante e mai avantajoasa sau nu.

4.1 Partea mecanica

Plotterul realizat pentru aceasta lucrare de diploma a fost realizat la **Facultatea de Mecanica**, cu ajutorul *domnului tehnician Traian Benea*. Proiectare lui a tinut cont de unele modele de plotter profesioniste sau "home-built", adica manufacturate acasa.

[1] limbajul BASIC avea comanda PLOT - punea puncte pe ecran

4.1.1 Functionare

In principiu, s-a dorit sa contina ca si componente sau comportare urmatoarele:

- desenare intr-un spatiu[2] 2D, in coordonate carteziene **X-Y**
- suport plan (SP) format A3 (maxim) pentru sustinerea celorlalte piese
- cate un motor pas cu pas pentru fiecare directie, unul fix (MF) si unul mobil (MM)
- suport (SC) pentru un creion cu tus (CT)
- releu (RL) care sa comande miscarea in cele doua pozitii ale CT (atinge da/nu foaia);
- RL, CT solidare cu SC
- surub de antrenare (SA) a SC, respectiv a MM
- ghidaje (GJ) pentru SC, respectiv MM

MF roteste solidar SA a MM, ghidat pe un GJ in plan orizontal, pe lungime; MM roteste solidar SA a SC, ghidat pe doua GJ in plan vertical, pe latime.
La extremitatea axului longitudinal (AL) este un lagar (LL); AL ramane fix.
La extremitatea axului transversal (AT) (care se misca ramanand perpendicular pe AL), este de asemenea un lagar (LT), care este inserat intr-un suport ce se misca paralel cu MM. Miscarea paralela ar fi putut fi asigurata prin culisarea LT, dar nesimetria intre culisarea LT si insurubarea MM, ar fi dus la blocarea miscarii longitudinale, datorita frecarii si a jocurilor.
Astfel ca au aparut noi elemente:

- suportul LT se insurubeaza pe un alt SA, paralel cu cel al MM; distanta dintre cele doua SA longitudinale (identice) este egala cu cea a SA transversal, iar miscarea lor trebuie sa fie simultana
- cel de-al doilea SA este prins in doua lagare (LL1, LL2), LL2 fiind in pozitie analoaga LL, dar pe alt SA
- MF are atasata o roata dintata (RD1), la fel si LL1 (RD2), avand aceleasi dimensiuni, solidare cu SA-urile lor; o curea cu dinti este prinsa pe circumferintele celor doua roti dintate, astfel incat orice miscare a axului MF se transmite si la LL1, astfel ca AT este deplasat fara blocare, "sprijinindu"-se pe cele doua SA longitudinale, paralele si sincronizate

Intrucat cureaua "facee burta" cand este pusa pe cele doua RD, s-au prevazut niste intinzatoare (IN) sub forma unor coroane fixate pe cate un rulment. Atat RD, cat si IN sunt prevazute cu discuri marginale ce nu lasa cureaua sa devieze de pe ele.

Odata stabilite aceste mecanisme de baza, s-au adaugat o serie de alte componente:

- s-au plasat pe suportul MM si pe cel al RL/CT cate doua microintrerupatoare (MI) de mouse; de asemenea pe cate unul dintre ghidajele paralele longitudinale, respectiv transversale s-au plasat niste opritoare (OP) ce culiseaza si se pot fixa rigid (cu un surub), manual sau cu surubelnita, ce comuta MI in starea *ON*, daca exista o deplasare care aduce coordonata curenta pe directia respectiva in zona limita, sau in starea *OFF*, daca exista o deplasare care readuce coordonata curenta in zona libera.

[2] bidimensional

- revenirea CT in pozitia "nu atinge HT" este sesizata prin comutarea unui MI; in acest moment este garantata lipsa contactului CT-HT; la comutarea inversa insa, garantarea contactului in momentul inceperii desenarii este conditionata de asteptarea epuizarii unui interval bine definit de timp (DT)[3]
- s-au plasat doua rigle esantionate (RE) in *cm* pe cele doua directii ortogonale, prinse cu suruburi cu arc, pentru rigidizarea pozitiei foii (HT) pe care se deseneaza, si pentru usurarea orientarii operatorului uman (OM) la plasarea OP.

Pozitia "nu atinge HT" a CT este una in care RL nu consuma curent (nu e alimentat); ratiunea a fost ca in majoritatea schitelor care pot fi desenate cu un plotter, densitatea "negrului" e mai mica decat a "albului", deci se economiseste putin din energia consumata. In aceasta pozitie, CT este "impins fortat in sus" printr-un arc (ARC1). Cand RL e cuplat, ARC1 este compresat de forta electromagnetica (FEM) generata de RL, care actioneaza asupra unei tije verticale solidare cu CT (cand acesta e constrans in suport), impingandu-l in jos. FEM este mult mai mare dacat forta elastica din ARC1 (FE1). In acest moment, CT este tinut rigid in contact cu HT/SP. Inerentele neplaneitati ale SP vor introduce niste forte de tensiune in CT, RL etc. care pot duce la reaua functionare a plotterului, ruperea hartiei, cine stie ce va ceda mai intai. De aceea, tija verticala a fost taiata in doua (!), intre cele doua tije formate punandu-se un arc (ARC2), a carui forta elastica (FE2) cand e compresat (deci in pozitia "atinge HT" a CT), e *mai mare* decat FE1, astfel ca se asigura contactul permanent CT-HT, permitand totodata preluarea "jocului" vertical datorat eventualelor neplaneitati.

4.1.2 Materiale

Majoritatea pieselor sunt din textolit, mai ales suportii.

Motoarele pas cu pas, precum si ghidajele sunt luate de la niste dispozitive date la casat, numite *iGraph*-uri, un gen de combinatie plotter-imprimanta:

- sunt fabricate de firma *MÆ*
- rezolutia este de $1,8\frac{o}{pas}$
- curentul per faza de $1A$
- rezistenta per faza de 5Ω

Suruburile de antrenare au:

- pasul de $1,5mm$
- diametrul de $\phi = 1.5cm$

4.2 Partea electromagnetica

4.2.1 Functionare

Miscarea pe cele doua axe este motorizata prin folosirea a 2 motoare pas cu pas identice, cu rezolutia de $1,8\frac{o}{pas}$ adica 200 de pasi pe rotatie. Motoarele sunt comandate printr-un

[3]TODO

circuit de putere [4] numit L298. Comanda logica vine de la pinii de date ai portului paralel, fiind intermediata printr-un registru latch 74LS373. Feedbackul de la microintrerupatoarele de limitare se stabilizeaza printr-un nivel de triggere Schmitt disponibile pe CI 74LS14, de unde se citesc, tot prin intermediul unui registru, prin pinii/bitii de stare ai portului paralel. Registrul de control al portului paralel se foloseste pentru (in)validarea alimentarii de putere a circuitelor L298, pentru sincronizarea incarcarii comenzilor fazelor la motoarele pas cu pas si pentru comanda releului ce comanda miscarea sus-jos a creionului.

Alimentarea logicii circuitelor se face cu 5V TTL; alimentarea de putere a motoarelor trebuie sa asigure $1A/\phi$ si avand in vedere ca rezistenta unei infasuratori a motorului este de 5Ω, rezulta ca ajunge 5V si pentru putere. Releul insa are o rezistenta de 80Ω, si asigura o forta satisfacatoare doar peste cel putin 12V. Asadar e de ajuns pentru alimentarea electronicii logice sau de putere o mufa de alimentare de la un calculator PC.

4.3 Driver Linux

Driverele[5] de dispozitiv in SO Linux se impart in doua categorii:

- orientate pe octet (CD)
- orientate pe bloc de octeti (BD)

Aceasta clasificare s-a facut din perspectiva interfetei procesor—periferic, sau software—hardware. Diferenta intre cele doua, la o prima intuitie, este ca la BD SO ofera zone de memorie pentru a acumula datele ce se vehiculeaza intre procesor—periferic, amanand transferul efectiv spre destinatiile utile pana in momentul umplerii tamponului sau a cererii explicite de “golire”, care apare prima data.
Bineinteles ca tipul driverului depinde in mare masura de afinitatea perifericului cu unul din cele doua moduri amintite.

4.3.1 Paradigma fisier-memorie-dispozitiv

Fisierele sunt un concept corelat cu *Sistemele de fisiere* (SF), care constituie un model de organizare a datelor. Alt concept este acela de directoare, sau cataloage, care sunt tot fisiere, dar destinate strict unei utilizari in cadrul SF, anume pentru mentinerea unei corespondente fisier logic (nume) — fisier fizic (date). Accesarea unui SF inseamna accesarea fisierelor lui, putandu-se obtine diverse informatii. O modalitate de acces este si citirea, respectiv scrierea in/din fisiere. La citire, datele unui fisier (ceea ce reprezinta ele), conteaza sa fie disponibile numai in momentul accesarii; procesorul lucrand direct numai cu memoria fizica si cu perifericele I/O, rezulta ca de oriunde ar veni datele, acestea sigur vor trece prin memorie sau prin periferice; astfel este posibil ca memoria si perifericele sa fie vazute tot ca fisiere.

In cazul memoriei un exemplu potrivit este ierarhia de directoare montata in subdirectorul */proc*, ale carui fisiere la citire ofera date despre starea sistemului de operare, colectate din structurile de date alocate in memorie la acel moment; scrierea (nu totdeauna posibila) poate fi o cale de modificare “la cald” a comportarii nucleului, deci a sistemelui.

In cazul perifericelor, daca este vorba de un controller de *unitate de disc dur* (HDD) — magnetic — sau de *unitate de disc optic* (CDROM) — optic —, datele procurate sunt

[4] de fapt doua, cate unul pentru fiecare axa
[5] software de conducere/control

asa zisele fisierele normale, adica date persistente pe care utilizatorul uman le manipuleaza in mod folositor — rolul sistemului de calcul.

Pentru ca aceste date sa fie transferate, este nevoie de un control si o de o supraveghere ale transferului, materializate tot printr-un transfer de date, dar am putea spune metadate, adica date a caror manipulare conduce la transferul corect al datelor folositoare. Pentru dispozitive a caror functionalitate nu implica existenta unor date utile, metadatele exista inca; efectul existentei lor in timp si spatiu poate fi intretinerea miscarii unui motor de curent continuu, sau a scanarii unei poze etc.

Directorul "/dev"

Rezulta ca notiunea de metadata exista ca entitate separata. Directorul */dev* are rolul asigurarii unui cadru standardizat de transfer al metadatelor.

Un fisier special din */dev*, de exemplu */dev/hda*, poate fi cel responsabil cu buna functionare a unui HDD.

Existenta numelor in ierarhia */dev* a dispozitivelor periferice, permite sa se faca distinctie intre acestea.

Modelul MAJOR/minor Modelul *Major/minor* (Mm) este motivat de impartirea logica a multimii dispozitivelor periferice pe categorii, diferentiat de tipul functionalitatii pe care o ofera. Astfel, diferite dispozitive ce difera ca functionalitate vor avea numere majore diferite; versiuni asemanatoare ale aceluiasi dispozitiv va avea numere minore diferite.

Un driver de dispozitiv este practic identificat de numarul major; el trebuie sa ofere suport pentru toate numerele minore (variante) existente.

In Linux, driverele de dispozitiv pot fi contactate prin intermediul directorului */dev*. In acest director, se gasesc asa numitele *fisiere speciale*, de tip caracter sau bloc. A accesa un asemenea fisier inseamna a contacta driverul corespunzator numarului major al fisierului. Daca s-a stabilit o conexiune, o aplicatie poate comunica in continuare cu dispozitivul prin operatii primitive *read, write, seek*, precum si prin operatii speciale de tip periferic *ioctl*s.

Dupa cum se stie, Linux are o viziune virtuala asupra sistemului de fisiere; astfel, fiecare subdirector este pasibil de a fi punct de montare a unor sisteme de fisiere native. O data montat, motorul sistemului de fisiere trebuie sa ofere posiblitata vizitarii cataloagelor si fisierelor lui, fiind indivizibil de sistemul virtual de fisiere; prin urmare, subsubdirectoarele acestuia pot fi de asemenea puncte de montare pentru alte sisteme de fisiere s.a.m.d. */dev* nu face exceptie, cazul lui particular de a fi si punct de contact cu dispozitivele ducand la o comportare interesanta.

Solutia "mknod" Mostenire destul de veche, modul de crearea a fisierelor speciale din subdirectorul */dev* este o solutie demodata si costisitoare din punct de vedere a spatiului de memorie pe harddisk. Distributiile Linux vin cu un program *MAKEDEV* care creeaza toate fisierele speciale deja consacrate. Este sarcina administratorului sa le instaleze, lucru ce necesita o experienta mai adanca in domeniul softwarelui low level.

O alta fateta nefavorabila este aceea ca sistemele de calcul folosesc o mica parte din directoarele */dev*, deoarece nu dispun de toate dispozitivele corespunzatoare; un sistem uzual are peste 1200 de noduri in */dev*.

Practic, este o modalitate *statica* de a rezolva o problema, si tot ce e "static" in Linux este supus demodarii si inlocuirii.

Solutia "devfs" *Device FileSystem* este o solutie inteligenta imaginata de Richard Gooch. In cazul anterior se poate observa ca aplicatiile se conecteaza la fiserele speciale uzitand numele acestora; e sarcina nucleului sa descopere numarul major asociat, pentru ca mai apoi sa ajunga la structura de memorie prin care acceseaza functiile de interfata ale driverului. Pe de alta parte, administratorul pleaca de la numerele majore si creeaza manual nume de fisiere pentru acestea; intamplator exista o conventie bine stabilita de asociere. Aceste conventii se stabilesc intre oamenii din domeniu.

devfs degreveaza partea umana de asemenea implicari manuale si inoportune in automatizarea sistemelor. Astfel, conventia de asociere intre nume de fisiere si numere majore nu mai e static, stabilita anterior, ci ea este alta pe fiecare sistem in parte; aceasta se poate deoarece fiecare driver isi *inregistreaza* un nume in sistemul de fisiere */dev* (un sistem virtual de data aceasta). Este oferi suport in continuare pentru numere majore si minore in sens clasic, deoarece exista aplicatii care lucreaza direct cu acestea. Totusi, numele unui driver ar ajunge pentru a-l identifica in mod unic.

Cu toate acestea, in aplicatia *plint* nu se foloseste aceasta solutie deoarece nu este inca standardizata in comunitatea Linux.

4.4 Ansamblu de module

Driverul este constituit din mai multe module:

device se ocupa cu interfatarea intre aplicatie si celelalte module ale driverului.

proc se ocupa cu afisarea starii reale a plotterului prin intermediul sistemului de fisiere virtual */proc*

primitives engine se ocupa cu transformarea primitivelor in comenzi de tip: forward, backward, hold. Sunt implementati algoritmi care sa optimizeze numarul de deplasari de baza.

smart RLE encoder preia comenzile de baza si le stocheaza intr-un buffer, intr-o forma codificata, beneficiind de o compresie RLE. Cand bufferul e plin, sau cand se doreste explicit, el este golit printr-o procedura de decodificare RLE si transmitere a comenzilor obtinute intr-o forma optima ultimului nivel software, cel care transpune efectiv comenzile soft in hard; nu se tine cont de rezolutie.

hardware interpretor preia o comand si un contor de repetitie, si tinand cont de rezolutie comanda dispozitivul hardware.

4.5 Pornire/Oprire

La initializarea driverului la momentul fazei de initializare a driverelor la pornirea sistemului, sau in momentul inserarii modulelor intr-un moment ulterior, se initializeaza driverul; aceasta presupune calibrarea lui, adica determinarea coordonatelor maxime pe ambele axe; calibrarea se face deplasand simultan CT *spre origine*, pana cand ambele limitatoare minime semnaleaza capatul, iar apoi in sens invers, *spre infinit*, pana cand ambele limitatoare maxime semnaleaza capatul. In acest moment se considera ca s-a calibrat hardwareul, si acesta intra in starea "sleep", in care nu se valideaza comanda motoarelor MF si MM si a releelor. Astfel, nu se va consuma energie electrica, fiind un mod economi de a "astepta" comenzi.

Cand “cineva” (aplicatia utilizator) doreste sa deseneze ceva, deschide fisierul */dev/plint*, iar in acest momentul hardwareul se “trezeste”, prin validarea comenzilor motoarelor si a releelor. Pe toata durata desenului comenzile raman active; totodata, daca driverul e sub forma de modul, el nu poate fi scos in acest moment.

La inchidere fisierului, hardwareul va “adormi” din nou, pana cand se va mai deschide din nou */dev/plint*. In timp ce e “doarme”, driverul poate fi scos, caz in care i se da sansa modulului hardware sa termine definitiv comanda.

4.6 Comunicarea /dev/plint

Fisierul */dev/plint* este locul comun unde aplicatia ce doreste sa deseneze se intalneste cu driverul. Prin simple apeluri sistem de tip *read, write, ioctl*, aplicate asupra acestui fisier, prin intermediul bufferelor argument, se comunica driverului primitivele ce se doresc a fi desenate. Prin intermediul asocierii unor numere fiecarei primitive, se efectueaza o codificare/decodificare binara a primitivelor. Odata decodificate la nivelul driverului, rezultatul e apelul unor interpretoare de primitive, care se afla in modulul “motor de primitive”.

4.7 Comunicarea /proc/driver/plint

Pentru fiecare ax se cunoaste rezolutia (cate puncte fizice — minimul de distanta— corespund la un punct logic), coordonata maxima, curenta si corespondenta in raport de numere intregi intre distanta fizica (in milimetrii) si numarul de puncte fizice. Toate aceste informatii pot fi vizualizate in spatiul utlizator prin citirea fisierului */proc/driver/plint*. Ele se iau in timp real din structurile de date ale driverului, folosindu-se “readw-write locks”.

4.8 Motorul de primitive

Motorul de primitive foloseste algoritmi in numere intregi pentru transformarea notiunilor de segmente intr-o serie de comenzi de baza, intr-o forma optima.

Pentru linii se foloseste algoritmul Bresenham, pentru cercuri si elipse algoritmul diferential cu octante.

4.9 Bufferul de comenzi

Existenta bufferul care tine stocate comenzile pana la golirea lor este motivat de aceea ca e primordiala asigurarea unui timp cat mai mic de executie a interpretorului. Pentru desene nu prea complicate, aceasta s-ar putea realiza daca se acumuleaza comenzile in buffer, interpretorul termina si desenarea merge in background. Tinand cont de faptul ca exista doar 3 comenzi de miscare: inainte, stai, inapoi, nu ar avea rost sa se tine pe 4 octeti un cod de miscare; e de ajuns 2 biti; ba mai mult s-ar putea ca sa se repete o serie de comenzi pe ambele axe de un numar de ori; se recurge prin urmare la o codificare RLE. Algoritmul e urmatorul:

```
un cod de miscare se (de)codifica astfel:
(01) i = 0; STARE = NORMAL;
(02) daca i == max treci la (09)
     altfel
     {
```

```
        citeste 2 (ab) biti de la indexul i din buffer
        ab | semnif.
        ---+--------
        00 | repaus
        01 | inainte
        11 | inapoi
        10 | special
        i += 2
      }
  (03) daca ab == 10 continua
       altfel reia de la (02)
  (04) citeste 2 biti (cd) de la indexul i din buffer
       cd | semnif.
       ---+--------
       00 | repaus
       01 | inainte
       11 | inapoi
       10 | creion
       i += 2
  (05) daca cd == 10
       {
          modifica starea de apasat a creionului
          reia de la (02)
       }
       altfel continua
  (06) STARE = RLE;
       citeste 16 biti (b15..b0) de la indexul i din buffer
       acestia reprezinta un numar pe 16 biti (n = 2^b15..b0)
       i += 16
  (07) pe parcursul urmatorelor n miscari executa miscarea
       codificata de cd (== 00, 01 sau 10) (i += m*2)
  (08) STARE = NORMAL; reia de la (02)
  (09) gata.

Nota: Avantajul apare cand sunt mai mult de 2 + PLOT_NR_COMMANDS
   comenzi succesive identice
```

4.10 Executia efectiva

Pentru intarzieri de ordinul sutelor de microsecunde si de ordinul milisecundelor se foloste ceasul in timp real (RTC) al calculatorului, la o frecventa dee 4096Hz. Pentru comandare RTC se foloste interfata driverului care vine odata cu nucleul de Linux. Din pacat, acesta nu ofera suport pentru aplicatii nucleu (cum este acest driver *plint*), ci doar pentru spatiul utilizator (adica nu exporta simboluri). Din fericire, s-au realizat patch-uri anume pentru comanda motoarelor care necesita timpi de acest ordin de intarziere.

Mai intai, pentru a folosi RTC procesul care print-un apel sistem sau intrerupere a ajuns in mod nucleu trebuie sa se inregistreze la driver. Doar un singur proces poate folosi driverul RTC la un moment dat (asa si driverul *plint*). Apoi, cand se doreste o anumita intarziere se activeaza intreruperile periodice (PIEnable) si se seteaza un contor in functie de raportul dintr 4096Hz si intarzierea dorita (calculul e foarte usor, doar prin niste shiftari pe biti); procesul intra apoi in “sleep”, pe o coada proprie de asteptare. La inregistrare s-a specificat o functie callback care e apelata la fiecare intrerupere de la RTC. Pe aceasta functie se decrementeaza contorul (bineinteles, in context utilizator altul decat al procesului de desenare, dar in mod nucleu, pe intrerupere); cand ajunge la 0, procesul se desteapta, folosind argumentul de pe stiva, care identifica coada de asteptare.

Beneficiind de acest tip de intarzieri, calculatorul va merge in paralel cu desenarea.

Fazele anterioara in care s-au aflat alimentarile motoarelor se memoreaza de la o deplasare la alta. Exista posibilitatea folosirii mai multor moduri de succesiune a fazelor. S-a ales modalitatea numita in semipasi. Caracteristicile plotterului sunt stocate intr-o structura de forma:

```
static struct
{

  char ver[4];
  int xadv, xstep, yadv, ystep; /* advance distance mm/1000 per number
      of steps */
  int no, phase[PLOT_AX_NO][16];
  unsigned long wait[4]; /* microseconds */
}
```

Cele folosite cu succes sunt:

```
  1500, 200, 1500, 200,
  8, {
    {1<<0,1<<0|1<<1,1<<1,1<<1|1<<2,1<<2,1<<2|1<<3,1<<3,1<<3|1<<0},
    {1<<4,1<<4|1<<5,1<<5,1<<5|1<<6,1<<6,1<<6|1<<7,1<<7,1<<7|1<<4}
  },
  { 64, 2048+1024, 1024*64, 1024*16 }
```

Odata stabilit plotterul curent printr-un pointer “p” la aceasta structura, modul de comanda a fazei urmatoare, functie de directie, uziteaza de urmatoarele macrouri:

```
typedef enum
{

  PLOT_CMD_BACKWARD  =  -1,
  PLOT_CMD_HOLD      =   0,
  PLOT_CMD_FOREWARD  =   1,
  PLOT_CMD_PENCIL    =   2,
  PLOT_CMD_SPECIAL   =   2,

} plot_cmd_t;

typedef enum
{

  PLOT_AX_S          = -1,
  PLOT_AX_X          = 0,
  PLOT_AX_Y          = 1,
  PLOT_AX_E          = 2,
  PLOT_AX_NO         = PLOT_AX_E,

#define foreachAx for (a = PLOT_AX_S; ++a < PLOT_AX_E ;)

} plot_ax_t;
```

```
plot_ax_t a;

static const int s_m[PLOT_AX_NO][2] = { {0x40, 0x80}, {0x20, 0x08} };
    /* status masks */
static const int s_b[PLOT_AX_NO][2] = { {0x00, 0x80}, {0x00, 0x00} };
    /* status bits */
static int i[PLOT_AX_NO];

#define NEXT(__dir__) i[a] += p->no + (__dir__), i[a] %= p->no

#define moveXY()            \
({                  \
  int h = 0, s = inb (STAT);       \
  foreachAx          \
    if ((s & s_m[a][d[a]]) == \
        (0 | s_b[a][d[a]])) { \
      h |= p->phase[a][i[a]]; \
      NEXT (PLOT_CMD(a)); \
    }         \
  phase (h);            \
})

#define moveAx()            \
({                  \
  int h = 0, s = inb (STAT);       \
  if ((s & s_m[a][d[a]]) ==     \
      (0 | s_b[a][d[a]])) {     \
    h |= p->phase[a][i[a]];     \
    NEXT (PLOT_CMD(a));     \
  }             \
  phase (h);            \
})
```

Capitolul 5

Final

Sursele de inspiratie sunt cursurile:

- Algebra liniara, prof. Dorina Rendi
- Calcul diferential, prof. Octavian Lipovan
- Matematici speciale, prof. Stefan Matei
- Calcul numeric, prof. Pavel Naslau
- Dispozitive Electronice, prof. Radu Boraci
- Structuri de Date si Algoritmi, prof. Vladimir Creţu
- Circuite Integrate, prof. Mircea Stratulat
- Calculatoare Numerice, prof. Mircea Vladuţiu
- Sisteme de operare, Ingineria Programarii, prof. Jurca Ioan
- Sistem Intrare Iesire, prof. Crisan Strugaru
- Sistem cu Microprocesoare, prof. Mircea Popa
- Proiectarea Translatoarelor/Tehnici de compilare, prof. Horia Ciocârile

De mare ajutor in crearea pluginului *class* este descrierea functionarii mecanismului oferit de C++ pentru definirea obiectelor, a modului de apel a metodelor virtuale ale claselor. Nu in ultimul rand inspiratie au mai fost si paginile de manual (man) si informatii (info) din SO Linux; apoi cartile (documentatiile) (electronice):

- The Linux Kernel, de David A. Rusling
- Linux Kernel Module Programming Guide, de Ori Pomerantz
- Linux Kernel Internals, de Tigran Aivazian tigran@veritas.com
- Filesystem hierarchy standard
- Linux Device Drivers, de Alessandro Rubini Jonathan Corbet, 2nd Edition June 2001

S-a folosit cod sursa din diverse aplicatii, printre care biblioteca *freetype*, designerul *dia*, translatorul *hp2xx*. Intreaga aplicatie a fost dezvoltata pe o platforma *Linux-based GNU i386*. Editorul de text folosit pentru dezvoltarea in C, LaTeX este *GNU Emacs*.
Pentru editarea shell-scripturilor si surselor *plint* s-a folosit editorul intern al aplicatiei *Midnight Commander*.
Acest document a fost creat in LaTeX.

Exista un proiect **Open Source** deschis la SourceForge, cu numele *plint*:

- pagina web: http://plint.sourceforge.net
- email administrator: Sebastian Glita
- acces CVS anonymous: -d z3 :pserver:plint.sourforge.net:/cvsroot/plint

Capitolul 6

Anexe

6.1 Lexicul

6.1.1 Lexicul miez

Clase lexicale

$\langle \mathit{WHITESPACESNL} \rangle$ ::= [$\backslash t$]

$\langle \mathit{WHITESPACES} \rangle$ ::= $\langle \mathit{WHITESPACESNL} \rangle$ | '$\backslash n$'

$\langle \mathit{LINECOMMENT} \rangle$ ::= //

$\langle \mathit{KW\_MISC} \rangle$ ::= plint|version|insert|include|echo|read|sync

$\langle \mathit{KW\_BLOCK} \rangle$ ::= _|0_|1_|2_|3_|4_|5_|6_|7_|8_|9_

$\langle \mathit{KW\_VAR} \rangle$ ::= new|type|at|var

$\langle \mathit{KW\_PROC} \rangle$ ::= figure|function|procedure|return|escape

$\langle \mathit{KW\_CTRL} \rangle$ ::= break|continue|loop|exit

$\langle \mathit{KW\_TYPE} \rangle$ ::= nul|number|integer|real|point|string|boolean|array|hash

$\langle \mathit{KW\_NUMBER} \rangle$ ::= crtx|crty|maxx|maxy|pntx|pnty

$\langle \mathit{KW\_POINT} \rangle$ ::= crtp|frame|P1|P2

$\langle \mathit{KW\_STRING} \rangle$ ::= this|plotter

$\langle \mathit{KW\_BOOLEAN} \rangle$ ::= true|false|last

$\langle \mathit{KW\_ARRAY} \rangle$ ::= argv|args

$\langle \mathit{KW\_VALUE} \rangle$ ::= $\langle \mathit{KW_NUMBER} \rangle$KW_POINT¿|$\langle \mathit{KW\_STRING} \rangle$|$\langle \mathit{KW\_BOOLEAN} \rangle$|$\langle \mathit{KW\_ARRAY} \rangle$

$\langle \mathit{KW\_OO} \rangle$ ::= method|self

⟨FN_PRIMITIVES⟩ ::= `ctrl`|`reset`|`select`|`unit`|`flush`
| `moveto`|`moverel`
| `lineto`|`linerel`|`line`
| `polylinerel`|`polyline`
| `squarerel`|`square`
| `rectanglerel`|`rectangle`
| `circlerel`|`circle`
| `ellipserel`|`ellipse`

⟨FN_NUMBER⟩ ::= `rnd`|`max`|`min`|`len`|`lev`|`fork`

⟨FN_VALUE⟩ ::= ⟨FN_NUMBER⟩

⟨SHORTOPS⟩ ::= `[-+*/%&|~^=<>‘]`

⟨PRFXOPS⟩ ::= `--`|`++`

⟨ASGNOPS⟩ ::= `+=`|`-=`|`*=`|`/=`|`%=`|`^=`|`&=`|`|=`|`~=`

⟨COMPOPS⟩ ::= `==`|`<=`|`=|=`|`>=`

⟨LOGIOPS⟩ ::= `&&`|`||`|`^^`|`~~`

⟨LONGOPS⟩ ::= ⟨PRFXOPS⟩|⟨ASGNOPS⟩|⟨COMPOPS⟩|⟨LOGIOPS⟩|`->`

⟨TABITEROPS⟩ ::= `[-+*/%&|~^=<>‘]`

⟨PUNCTUATORS⟩ ::= `[`\[\]–”() :; , .\

⟨NUMSPEC⟩ ::= `$`

⟨PNTSPEC⟩ ::= `@`

⟨BOLSPEC⟩ ::= `?`

⟨STRSPEC⟩ ::= `"`

⟨TABSPEC⟩ ::= `!`

⟨HSHSPEC⟩ ::= `#`

⟨SPECIFICATORS⟩ ::= ⟨NUMSPEC⟩|⟨PNTSPEC⟩|⟨TABSPEC⟩|⟨BOLSPEC⟩|⟨STRSPEC⟩|⟨HSHSPEC⟩

⟨SEPARATORS⟩ ::= ⟨WHITESPACES⟩|⟨SHORTOPS⟩|⟨LONGOPS⟩|⟨PUNCTUATORS⟩

⟨NZDECDIGIT⟩ ::= `[1-9]`

⟨DECDIGIT⟩ ::= `[0-9]`

⟨NOTDECDIGIT⟩ ::= `[^0-9]`

⟨STARTDECDIGIT⟩ ::= `0[^xX_]`|⟨NZDECDIGIT⟩`[^_]`|`[.]`⟨DECDIGIT⟩

⟨*DECINT*⟩ ::= (0+|0*⟨*NZDECDIGIT*⟩⟨*DECDIGIT*⟩{0,9})[^.]

⟨*BADDECINT*⟩ ::= 0*⟨*NZDECDIGIT*⟩⟨*DECDIGIT*⟩+[^.]

⟨*DECDBLINT*⟩ ::= 0+|0*⟨*NZDECDIGIT*⟩⟨*DECDIGIT*⟩{0,9}

⟨*BADDECDBLINT*⟩ ::= 0*⟨*NZDECDIGIT*⟩⟨*DECDIGIT*⟩+

⟨*DECDBLFRAC*⟩ ::= ⟨*DECDIGIT*⟩{1,20}⟨*NOTDECDIGIT*⟩

⟨*BADDECDBLFRAC*⟩ ::= ⟨*DECDIGIT*⟩+

⟨*DECDBL*⟩ ::= ⟨*DECDBLINT*⟩[.]⟨*NOTDECDIGIT*⟩|⟨*DECDBLINT*⟩?[.]⟨*DECDBLFRAC*⟩

⟨*BADDECDBL*⟩ ::= ⟨*BADDECDBLINT*⟩[.]|⟨*DECDBLINT*⟩?[.]⟨*BADDECDBLFRAC*⟩

⟨*NZHEXDIGIT*⟩ ::= [1-9a-fA-F]

⟨*HEXDIGIT*⟩ ::= [0-9a-fA-F]

⟨*NOTHEXDIGIT*⟩ ::= [^0-9a-fA-F]

⟨*STARTHEXDIGIT*⟩ ::= 0[xX]

⟨*HEXINT*⟩ ::= 0+|0*⟨*NZHEXDIGIT*⟩⟨*HEXDIGIT*⟩{0,7}

⟨*BADHEXINT*⟩ ::= 0*⟨*NZHEXDIGIT*⟩⟨*HEXDIGIT*⟩+

⟨*DIGITS*⟩ ::= [0-9]

⟨*LETTERS*⟩ ::= [a-zA-Z]

⟨*IDENT1ST*⟩ ::= ⟨*LETTERS*⟩|_

⟨*IDENTRST*⟩ ::= ⟨*LETTERS*⟩|⟨*DIGITS*⟩|_

⟨*IDENTIFIER*⟩ ::= ⟨*IDENT1ST*⟩ ⟨*IDENTRST*⟩{0,128}

⟨*BADIDENTIFIER*⟩ ::= ⟨*IDENT1ST*⟩ ⟨*IDENTRST*⟩+

⟨*STRING*⟩ ::= [- 0-9a-zA-Z~‘!@#$%^&*()_+|=\[\]{};\" :,./ <>?]

⟨*NOTSTRING*⟩ ::= [^- 0-9a-zA-Z~‘!@#$%^&*()_+|=\\\[\]{};\" :,./ <>?\t\r\b\n\a\f\v]

Actiuni lexicale

```
        /* start conditions (or states) */

\%x comment

\%x expression
\%x dec
\%x hex
```

```
\%x string


<INITIAL><<EOF>>                       {
  switch (plint_run)
    {
    case 0:return plint_parse_eof ();
    case 1:return plint_run_eof ();
    case 2:return 0; /* done running top block */
    }
}

<<EOF>>                                        PLINT_ERROR (PLINT_ERR_EOF);


        /* whitespaces */
[\n]+

{WHITESPACESNL}+                       /* discard */

        /* shell autorun feature */
^#![^\n]*\n


        /* keywords */
{KEYWORDS}/{SEPARATORS}|{STRSPEC}


        /* functions */
{FUNCTIONS}/{WHITESPACES}*[(]


        /* identifiers */
{IDENTIFIER}/{SEPARATORS}|{STRSPEC}|{NUMSPEC}


{BADIDENTIFIER}                                PLINT_ERROR (PLINT_ERR_IDENT);


        /* long operators */
{PRFXOPS}|{ASGNOPS}|"->"|{COMPOPS}|{LOGIOPS}

        /* short operators */
{SHORTOPS}

        /* a number can be in decimal or in hex */
{STARTHEXDIGIT}                                { num_base = 16; BEGIN (hex); }
```

```
{STARTDECDIGIT}                              { num_base = 10; BEGIN (dec); }

        /* a decimal number (long) */
<dec>{DECINT}                              { num_ok (INTEGER); BEGIN (INITIAL); }

<dec>{BADDECINT}                        PLINT_ERROR (PLINT_ERR_NUMBER);

        /* a decimal number (double) */
<dec>{DECDBL}                              { num_ok (DOUBLE); BEGIN (INITIAL); }

<dec>{BADDECDBL}                        PLINT_ERROR (PLINT_ERR_NUMBER);

        /* a hex number */
<hex>{HEXINT}                              { num_ok (INTEGER); BEGIN (INITIAL); }

<hex>{BADHEXINT}                        PLINT_ERROR (PLINT_ERR_NUMBER);


        /* a line or block comment */
^{WHITESPACESNL}*{LINECOMMENT}.*\n

"/*"                               BEGIN (comment);
<comment>[\n]+
<comment>[^*\n]*                /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*                /* eat up '*'s not followed by '/'s */
<comment>"*"+"/"                BEGIN (INITIAL);


        /* a string */
\'                                  { BEGIN (string); }
<string>[\n]+                          /* ignore */
<string>[\t\r\b]+                   /* ignore */
<string>\'
<string>\\backslash
<string>\h
<string>\b
<string>\t
<string>\n
<string>\v
<string>\f
<string>\r
<string>\.
<string>{NOTSTRING}+
<string>{STRING}+
<string>\'                          { BEGIN (INITIAL); }

        /* specificators */
{SPECIFICATORS}
```

```
        /* meta characters */
\.

        /* character tokens (or last-tried,
           bad tokens for various states) */
<*>.
```

6.2 Sintaxa BNF

6.2.1 Sintaxa miez

Un program sursa *plint* consta din doua parti: antet si procedura principala. Antetul are rolul facilitarii executiei fisierului din shell[1], de precizare a versiunii programului *plint* cu care a fost ultima oara testat bine fisierul, precum si *plint* plotterele pentru care a fost mai ales conceput; aceste ultime doua pot lipsi.

Programul principal, bloc, sectiuni

⟨*interpret*⟩ ::= ⟨*header*⟩ ⟨*block*⟩

⟨*header*⟩ ::= ⟨*autorun*⟩ ⟨*version*⟩ ⟨*plotters*⟩ `plint` `STRING`

⟨*autorun*⟩ ::= ⟨*empty*⟩ | `autorun`

⟨*version*⟩ ::= ⟨*empty*⟩ | `version` `INTEGER` ‘`,`’ `INTEGER`

⟨*plotters*⟩ ::= ⟨*empty*⟩ | `plotter` pltlist

⟨*pltlist*⟩ ::= `STRING` | pltlist ‘`,`’ `STRING`

⟨*block*⟩ ::= ‘`{`’ blackhole ‘`}`’

⟨*blackhole*⟩ ::= ⟨*empty*⟩ | ⟨*blackhole*⟩ ⟨*section*⟩

⟨*section*⟩ ::= ⟨*structure*⟩ | ⟨*data*⟩ | ⟨*flow*⟩ | ⟨*io*⟩ | ⟨*plugins_section*⟩

Structura

⟨*structure*⟩ ::= ⟨*files*⟩ | ⟨*blocks*⟩ | ⟨*plugins_structure*⟩

⟨*files*⟩ ::= ⟨*embed*⟩

⟨*blocks*⟩ ::= ⟨*named*⟩ | ⟨*anonymous*⟩

⟨*named*⟩ ::= ⟨*define*⟩

⟨*anonymous*⟩ ::= ⟨*block*⟩

⟨*embed*⟩ ::= ⟨*insinc*⟩ `STRING` ⟨*blackhole*⟩

[1] shell = program de lansare a altor procese

⟨*define*⟩ ::= ⟨*procmet*⟩ `IDENTIFIER` ‘(’ ⟨*namelist*⟩ ‘)’ ⟨*hook*⟩ ⟨*block*⟩

⟨*procmet*⟩ ::= `procedure` | `method`

⟨*namelist*⟩ ::= ⟨*empty*⟩ | namelist1

⟨*namelist1*⟩ ::= ⟨*nameitem*⟩ | ⟨*namelist*⟩ ‘,’ ⟨*nameitem*⟩

⟨*nameitem*⟩ ::= `IDENTIFIER`

⟨*hook*⟩ ::= ⟨*plugins_hook*⟩

Date

⟨*data*⟩ ::= ⟨*variables*⟩ | ⟨*plugins_data*⟩

⟨*variables*⟩ ::= ⟨*new*⟩ | ⟨*assignment*⟩ | ⟨*operasg*⟩

Creare

⟨*new*⟩ ::= ‘‘’ ⟨*type*⟩ ⟨*newlist*⟩ ‘;’ ⟨*iterate*⟩

⟨*newlist*⟩ ::= ⟨*newitem*⟩ | ⟨*newlist*⟩ ‘,’ ⟨*newitem*⟩

⟨*newitem*⟩ ::= ⟨*newididxs*⟩ | ⟨*newididxs*⟩ ⟨*args*⟩

⟨*newididxs*⟩ ::= ⟨*identifier*⟩ ⟨*idxlist*⟩

Asignare

⟨*assignment*⟩ ::= ⟨*asgnlist*⟩ ⟨*assignment1*⟩

⟨*assignment1*⟩ ::= ‘=’ ⟨*asgnsrc*⟩ ‘;’ ⟨*iterate*⟩
| ⟨*asgoper*⟩ ⟨*rvalue*⟩ ‘;’ ⟨*iterate*⟩

⟨*asgnlist*⟩ ::= ⟨*asgnitem*⟩ | ⟨*asgnlist*⟩ ‘,’ ⟨*asgnitem*⟩

⟨*asgnitem*⟩ ::= ⟨*asgndst*⟩

⟨*asgndst*⟩ ::= ⟨*lvalue*⟩
| ‘=’ ⟨*lvalue*⟩

⟨*asgnsrc*⟩ ::= ⟨*nulvar*⟩
| ⟨*rvalue*⟩

⟨*operasg*⟩ ::= ⟨*asgoper*⟩ ‘(’ ⟨*reference*⟩ ‘,’ ⟨*tabvar*⟩ ‘)’ ⟨*iterate*⟩

Flux

⟨*flow*⟩ ::= ‘;’ | ⟨*primitives*⟩ | ⟨*control*⟩ | ⟨*sideeffect*⟩ | ⟨*plugins_flow*⟩

Primitive (proceduri C)

⟨*primitives*⟩ ::= `reset` ‘(’ ‘)’
| `flush` ‘(’ ‘)’
| `select` ‘(’ ⟨*numvar*⟩ ‘)’
| `moveto` ‘(’ ⟨*pntarg*⟩ ‘)’
| `moverel` ‘(’ ⟨*pntarg*⟩ ‘)’
| `line` ‘(’ ⟨*zptarg*⟩ ‘)’
| `lineto` ‘(’ ⟨*pntarg*⟩ ‘)’
| `linerel` ‘(’ ⟨*pntarg*⟩ ‘)’
| `square` ‘(’ ⟨*pntarg*⟩ ‘,’ ⟨*numvar*⟩ ‘)’
| `squarerel` ‘(’ ⟨*numvar*⟩ ‘)’
| `rectangle` (’ ⟨*zptarg*⟩ ‘)’
| `rectanglerel` ‘(’ ⟨*numvar*⟩ ‘,’ ⟨*numvar*⟩ ‘)’
| `circle` ‘(’ ⟨*pntarg*⟩ ‘,’ ⟨*numvar*⟩ ‘)’
| `circle` (’ ⟨*pntarg*⟩ ‘,’ ⟨*numvar*⟩ ‘,’ ⟨*numvar*⟩ ‘)’
| `circlerel` ‘(’ ⟨*numvar*⟩ ‘)’
| `circlerel` ‘(’ ⟨*numvar*⟩ ‘,’ ⟨*numvar*⟩ ‘)’
| `ellipse` ‘(’ ⟨*pntarg*⟩ ‘,’ ⟨*numvar*⟩ ‘,’ ⟨*numvar*⟩ ‘)’
| `ellipse` ‘(’ ⟨*pntarg*⟩ ‘,’ ⟨*numvar*⟩ ‘,’ ⟨*numvar*⟩ ‘,’ ⟨*numvar*⟩ ‘)’
| `ellipserel` ‘(’ ⟨*numvar*⟩ ‘,’ ⟨*numvar*⟩ ‘)’
| `ellipserel` ‘(’ ⟨*numvar*⟩ ‘,’ ⟨*numvar*⟩ ‘,’ ⟨*numvar*⟩ ‘)’
| `polyline` ‘(’ ⟨*tabvar*⟩ ‘)’
| `polylinerel` ‘(’ ⟨*tabvar*⟩ ‘)’
| ⟨*plugins_primitives*⟩

Efecte laterale

⟨*sideffect*⟩ ::= ⟨*call*⟩ ‘;’ | ⟨*idsevar*⟩ ‘;’

⟨*call*⟩ ::= ⟨*procname*⟩ ⟨*args*⟩
| ⟨*methname*⟩ ⟨*args*⟩
| ⟨*methcall*⟩ ⟨*args*⟩

⟨*procname*⟩ ::= ⟨*identifier*⟩

⟨*methname*⟩ ::= ⟨*xxxvar*⟩ ‘‘’ ⟨*identifier*⟩ ⟨*action*⟩

⟨*methcall*⟩ ::= ⟨*xxxvar*⟩ ‘‘’

Controlul fluxului

⟨*ctlcnt*⟩ ::= ⟨*empty*⟩ | ⟨*numvar*⟩

⟨*control*⟩ ::= ⟨*blockctrl*⟩ | ⟨*procctrl*⟩

⟨*procctrl*⟩ ::= ⟨*return*⟩ | ⟨*escape*⟩ | ⟨*exit*⟩

⟨*exit*⟩ ::= `exit`

⟨*return*⟩ ::= `return` ⟨*ctlcnt*⟩ ‘;’

⟨*escape*⟩ ::= `escape` ⟨*ctlcnt*⟩ ‘`;`’

⟨*blockctrl*⟩ ::= ⟨*break*⟩ | ⟨*continue*⟩ | ⟨*contloop*⟩

⟨*break*⟩ ::= `break` ⟨*ctlcnt*⟩ ‘`;`’

⟨*continue*⟩ ::= `continue` ⟨*ctlcnt*⟩ ‘`;`’

⟨*contloop*⟩ ::= `continue loop` ⟨*ctlcnt*⟩ ‘`;`’

Intrare/Iesire

⟨*io*⟩ ::= `echo version` ‘`;`’ | ⟨*eco*⟩ | ⟨*sync*⟩

⟨*eco*⟩ ::= `echo` ⟨*ecolist*⟩ ‘`;`’

⟨*ecolist*⟩ ::= ⟨*ecoelist*⟩ | ⟨*ecolist*⟩ ‘`,`’ ⟨*ecoelist*⟩

⟨*ecoelist*⟩ ::= ⟨*nulvar*⟩ | ⟨*rvalue*⟩

⟨*sync*⟩ ::= `sync` ⟨*synclist*⟩ ‘`;`’

⟨*synclist*⟩ ::= ⟨*syncelist*⟩ | ⟨*synclist*⟩ ‘`,`’ ⟨*syncelist*⟩

⟨*syncelist*⟩ ::= ⟨*rvalue*⟩

Reguli ajutatoare

⟨*args*⟩ ::= ‘`(`’ ⟨*tablist*⟩ ⟨*maketab*⟩ ‘`)`’
| ‘`!`’ ‘`(`’ ⟨*tabvar*⟩ ‘`)`’

⟨*pntarg*⟩ ::= ⟨*point*⟩ | ⟨*number*⟩ ‘`,`’ ⟨*number*⟩

⟨*zptarg*⟩ ::= ⟨*zpoint*⟩ | ⟨*pntarg*⟩ ‘`,`’ ⟨*pntarg*⟩

⟨*asgoper*⟩ ::= `+=` | `-=` | `*=` | `/=` | `%=` | `^=` | `&=` | `|=` | `~=`

⟨*fncoper*⟩ ::= ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’ | ‘`%`’ | ‘`^`’ | ‘`&`’ | ‘`|`’ | ‘`~`’

⟨*ppmmoper*⟩ ::= `++` | `--`

⟨*cmpoper*⟩ ::= ‘`<`’ | ‘`>`’ | `<=` | `>=` | `==` | `=|=`

⟨*cnvoper*⟩ ::= ‘`$`’ | ‘`@`’ | ‘`"`’ | ‘`?`’ | ‘`!`’ | ‘`#`’

⟨*rmoper*⟩ ::= ‘`,`’ | ‘`:`’

⟨*metaoper*⟩ ::= `new` | `echo` | `read` | `sync` | ‘`<`’ ‘`>`’ | ‘`(`’ ‘`)`’

⟨*lvalue*⟩ ::= ⟨*reference*⟩
| ⟨*plugins_lvalues*⟩

⟨*rvalue*⟩ ::= ⟨*blkvar*⟩
  | ⟨*rnew*⟩ ⟨*identifier*⟩ ⟨*args*⟩ ⟨*action*⟩ ⟨*idxlist*⟩
  | ⟨*plugins_rvalues*⟩

⟨*rnew*⟩ ::= `new` | ‘’

⟨*identifier*⟩ ::= `IDENTIFIER`
  | `this` | `self`
  | ‘"’ ⟨*strconv*⟩ ‘"’
  | ‘<’ ⟨*string*⟩ ‘>’
  | ⟨*plugins_identifiers*⟩

⟨*valtype*⟩ ::= `nul` | `number` | `integer` | `double` | `point` | `string` | `boolean` | `table` | `hash`
  | `type` ‘"’ ⟨*strconv*⟩ ‘"’
  | `type` ‘<’ ⟨*string*⟩ ‘>’
  | `type` ‘(’ ⟨*blkvar*⟩ ‘)’

⟨*vartype*⟩ ::= ‘(’ `type` ‘)’ ⟨*variable*⟩

⟨*type*⟩ ::= ⟨*identifier*⟩
  | ⟨*vartype*⟩

⟨*cast*⟩ ::= ‘<’ ⟨*type*⟩ ‘>’

⟨*at*⟩ ::= ⟨*empty*⟩
  | `at` ⟨*numvar*⟩
  | `at last`
  | `at last` ⟨*numvar*⟩

⟨*id*⟩ ::= ⟨*identifier*⟩ | `argv`
  | ‘_’ | `0_` | `1_` | `2_` | `3_` | `4_` | `5_` | `6_` | `7_` | `8_` | `9_`

⟨*idxlist*⟩ ::= ⟨*empty*⟩ | ⟨*idxlist*⟩ ⟨*index*⟩

⟨*index*⟩ ::= ‘$’ ⟨*id*⟩
  | ‘[’ ⟨*numvar*⟩ ‘]’
  | ‘.’ ⟨*identifier*⟩

⟨*rangelist*⟩ ::= ⟨*rangeitem*⟩ | ⟨*rangelist*⟩ ‘:’ ⟨*rangeitem*⟩

⟨*rangeitem*⟩ ::= ⟨*empty*⟩
  | ⟨*number*⟩
  | ⟨*number*⟩ ‘,’
  | ‘,’ ⟨*number*⟩
  | ⟨*number*⟩ ‘,’ ⟨*number*⟩
  | ‘,’

Referinte

⟨*reference*⟩ ::= ⟨*idref*⟩ | ⟨*callref*⟩

⟨*idref*⟩ ::= ⟨*id*⟩ ⟨*refindex*⟩

⟨*callref*⟩ ::= ⟨*call*⟩ ⟨*refindex*⟩

⟨*refindex*⟩ ::= ⟨*empty*⟩ | ⟨*refindex*⟩ ⟨*indexref*⟩

⟨*indexref*⟩ ::= ’$’ ⟨*id*⟩
| ‘[’ ⟨*number*⟩ ‘]’
| ‘.’ ⟨*identifier*⟩
| ‘[’ ⟨*string*⟩ ‘]’

Variabile

⟨*refvar*⟩ ::= ⟨*reference*⟩ | ⟨*castvar*⟩

⟨*castvar*⟩ ::= ⟨*cast*⟩ ⟨*idref*⟩ ⟨*action*⟩
| ⟨*cast*⟩ ‘(’ ⟨*blkvar*⟩ ‘)’ ⟨*action*⟩
| ⟨*cast*⟩ ⟨*castvar*⟩ ⟨*action*⟩

⟨*idsevar*⟩ ::= ⟨*incdecvar*⟩ ⟨*action*⟩

⟨*incdecvar*⟩ ::= ⟨*ppmmoper*⟩ ⟨*reference*⟩

⟨*bltinvar*⟩ ::= ⟨*refvar*⟩ | ⟨*idsevar*⟩ | ‘(’ ‘)’

⟨*opervar*⟩ ::= ⟨*variable*⟩ ⟨*fncoper*⟩ ⟨*variable*⟩
| ⟨*fncoper*⟩ ⟨*variable*⟩
| ⟨*variable*⟩ ⟨*ppmmoper*⟩
| ⟨*variable*⟩ ‘{’ ⟨*rangelist*⟩ ‘}’ ⟨*iterate*⟩

⟨*basicvar*⟩ ::= ⟨*opervar*⟩ ⟨*action*⟩

⟨*mixtvar*⟩ ::= `read` ⟨*type*⟩ ⟨*action*⟩
| ‘[’ ⟨*tabvar*⟩ ‘]’ ⟨*number*⟩

⟨*textvar*⟩ ::= ‘(’ ⟨*blkvar*⟩ ‘)’
| ⟨*bltinvar*⟩
| ⟨*basicvar*⟩
| ⟨*mixtvar*⟩
| ⟨*plugins_var*⟩

⟨*variable*⟩ ::= ⟨*textvar*⟩

⟨*xxxvar*⟩ ::= ⟨*refvar*⟩ | ⟨*idsevar*⟩

⟨*blkvar*⟩ ::= ⟨*block*⟩ | ⟨*variable*⟩

⟨*nulvar*⟩ ::= ⟨*empty*⟩

⟨*cmpvar*⟩ ::= ⟨*variable*⟩ ⟨*cmpoper*⟩ ⟨*variable*⟩

Numere intregi/reale

```
⟨bltinnum⟩   ::= INTEGER | DOUBLE
              |  len '(' ⟨blkvar⟩ ')'
              |  len '(' ⟨number⟩ ')'
              |  len '(' ⟨point⟩ ')'
              |  len '(' ⟨string⟩ ')'
              |  len '(' ⟨boolean⟩ ')'
              |  len '(' ⟨table⟩ ')'
              |  len '(' ⟨hash⟩ ')'
              |  rnd '(' ⟨numvar⟩ ')'
              |  min '(' ⟨numvar⟩ ',' ⟨numvar⟩ ')'
              |  max '(' ⟨numvar⟩ ',' ⟨numvar⟩ ')'
              |  fork '(' ')'
              |  lev '(' ')'
              |  lev '(' ⟨numvar⟩ ')'

⟨basicnum⟩   ::= ⟨number⟩ '+' ⟨number⟩
              |  ⟨number⟩ '-' ⟨number⟩
              |  ⟨number⟩ '*' ⟨number⟩
              |  ⟨number⟩ '/' ⟨number⟩
              |  ⟨number⟩ '%' ⟨number⟩
              |  ⟨number⟩ '&' ⟨number⟩
              |  ⟨number⟩ '|' ⟨number⟩
              |  ⟨number⟩ '^' ⟨number⟩
              |  '+' ⟨number⟩
              |  '-' ⟨number⟩
              |  '/' ⟨number⟩
              |  '~' ⟨number⟩

⟨mixtnum⟩    ::= '*' ⟨string⟩
              |  crtx | crty | maxx | maxy
              |  pntx '(' ⟨pntvar⟩ ')'
              |  pnty '(' ⟨pntvar⟩ ')'

⟨textnum⟩    ::= '(' ⟨number⟩ ')'
              |  ⟨bltinnum⟩
              |  ⟨basicnum⟩
              |  ⟨mixtnum⟩
              |  ⟨plugins_num⟩

⟨number⟩     ::= ⟨textnum⟩
              |  '$' ⟨numconv⟩

⟨numconv⟩    ::= ⟨numconvar⟩ ⟨action⟩

⟨numconvar⟩  ::= ⟨numblkvar⟩ | ⟨numxxxvar⟩

⟨numblkvar⟩  ::= '(' ⟨rvalue⟩ ')'
```

⟨*numxxxvar*⟩ ::= ⟨*xxxvar*⟩

⟨*numvar*⟩ ::= ⟨*number*⟩ | ⟨*blkvar*⟩

Puncte 2D

⟨*bltinpnt*⟩ ::= `crtp`
| ‘`(`’ ⟨*number*⟩ ‘`,`’ ⟨*number*⟩ ‘`)`’
| ‘`[`’ ⟨*number*⟩ ‘`,`’ ⟨*number*⟩ ‘`]`’

⟨*basicpnt*⟩ ::= ‘`-`’ ⟨*point*⟩
| ‘`/`’ ⟨*point*⟩
| ⟨*point*⟩ ‘`+`’ ⟨*point*⟩
| ⟨*point*⟩ ‘`-`’ ⟨*point*⟩
| ⟨*point*⟩ ‘`*`’ ⟨*point*⟩

⟨*mixtpnt*⟩ ::= ⟨*zptpnt*⟩
| ⟨*number*⟩ ‘`*`’ ⟨*point*⟩
| ⟨*point*⟩ ‘`+`’ ⟨*number*⟩
| ⟨*point*⟩ ‘`-`’ ⟨*number*⟩
| ⟨*point*⟩ ‘`*`’ ⟨*number*⟩
| ⟨*point*⟩ ‘`/`’ ⟨*number*⟩
| ⟨*point*⟩ ‘`%`’ ⟨*number*⟩

⟨*zptpnt*⟩ ::= `P1` ‘`(`’ ⟨*zpoint*⟩ ‘`)`’ | `P2` ‘`(`’ ⟨*zpoint*⟩ ‘`)`’

⟨*textpnt*⟩ ::= ‘`(`’ ⟨*point*⟩ ‘`)`’
| ⟨*bltinpnt*⟩
| ⟨*basicpnt*⟩
| ⟨*mixtpnt*⟩
| ⟨*plugins_pnt*⟩

⟨*point*⟩ ::= ⟨*textpnt*⟩
| ‘`@`’ ⟨*pntconv*⟩

⟨*pntconv*⟩ ::= ⟨*pntconvar*⟩ ⟨*action*⟩

⟨*pntconvar*⟩ ::= ⟨*pntblkvar*⟩ | ⟨*pntxxxvar*⟩

⟨*pntblkvar*⟩ ::= ‘`(`’ ⟨*rvalue*⟩ ‘`)`’

⟨*pntxxxvar*⟩ ::= ⟨*xxxvar*⟩

⟨*pntvar*⟩ ::= ⟨*point*⟩ | ⟨*blkvar*⟩

⟨*bltin2pt*⟩ ::= `frame` | ‘`[`’ ⟨*point*⟩ ‘`,`’ ⟨*point*⟩ ‘`]`’ | ‘`(`’ ⟨*point*⟩ ‘`,`’ ⟨*point*⟩ ‘`)`’

⟨*text2pt*⟩ ::= ⟨*bltin2pt*⟩ | ⟨*plugins_2pt*⟩

⟨*Zpoint*⟩ ::= ⟨*text2pt*⟩

Siruri de caractere

⟨*bltinstr*⟩ ::= `STRING` | `plotter`

⟨*basicstr*⟩ ::= ⟨*string*⟩ ‘`+`’ ⟨*string*⟩
| ⟨*string*⟩ ‘`-`’ ⟨*string*⟩
| ⟨*string*⟩ ‘`%`’ ⟨*string*⟩
| ⟨*string*⟩ ‘`~`’ ⟨*string*⟩
| ‘`+`’ ⟨*string*⟩
| ‘`-`’ ⟨*string*⟩
| ‘`/`’ ⟨*string*⟩
| ⟨*string*⟩ ‘`{`’ rangelist ‘`}`’
| ⟨*string*⟩ ‘`/`’ ⟨*string*⟩ ‘`/`’ ⟨*string*⟩

⟨*mixtstr*⟩ ::= ⟨*string*⟩ ‘`*`’ ⟨*number*⟩
| ‘`"`’ ⟨*identifier*⟩ ‘`"`’
| ‘`"`’ `type` ‘`(`’ ⟨*variable*⟩ ‘`)`’

⟨*textstr*⟩ ::= ‘`(`’ ⟨*string*⟩ ‘`)`’
| ⟨*basicstr*⟩
| ⟨*bltinstr*⟩
| ⟨*mixtstr*⟩
| ⟨*plugins_str*⟩

⟨*string*⟩ ::= ⟨*textstr*⟩
| ‘`"`’ ⟨*strconv*⟩

⟨*strconv*⟩ ::= ⟨*strconvar*⟩ ⟨*action*⟩

⟨*strconvar*⟩ ::= ⟨*strblkvar*⟩ | ⟨*strxxxvar*⟩

⟨*strblkvar*⟩ ::= ‘`(`’ ⟨*rvalue*⟩ ‘`)`’

⟨*strxxxvar*⟩ ::= ⟨*xxxvar*⟩

⟨*strvar*⟩ ::= ⟨*string*⟩ | ⟨*blkvar*⟩

Booleeni - adevarat/fals

⟨*bltinbol*⟩ ::= `true` | `false`
| `last`
| `var` ⟨*id*⟩ | `var` ‘`(`’ ⟨*id*⟩ ⟨*at*⟩ ‘`)`’

⟨*basicbol*⟩ ::= `~~` ⟨*boolean*⟩
| ⟨*boolean*⟩ `&&` ⟨*boolean*⟩
| ⟨*boolean*⟩ `||` ⟨*boolean*⟩
| ⟨*boolean*⟩ `^^` ⟨*boolean*⟩
| ⟨*boolean*⟩ `==` ⟨*boolean*⟩
| ⟨*boolean*⟩ `=|=` ⟨*boolean*⟩
| ⟨*boolean*⟩ `<=` ⟨*boolean*⟩
| ⟨*boolean*⟩ `>=` ⟨*boolean*⟩

⟨*mixtbol*⟩ ::= ⟨*cmpvar*⟩ ⟨*action*⟩
| ⟨*valtype*⟩ `==` ⟨*valtype*⟩
| ⟨*valtype*⟩ `=|=` ⟨*valtype*⟩
| ⟨*cast*⟩ `==` ⟨*cast*⟩
| ⟨*cast*⟩ `=|=` ⟨*cast*⟩
| ⟨*number*⟩ `==` ⟨*number*⟩
| ⟨*number*⟩ `=|=` ⟨*number*⟩
| ⟨*number*⟩ ‘`<`’ ⟨*number*⟩
| ⟨*number*⟩ ‘`>`’ ⟨*number*⟩
| ⟨*number*⟩ `<=` ⟨*number*⟩
| ⟨*number*⟩ `>=` ⟨*number*⟩
| ⟨*point*⟩ `==` ⟨*point*⟩
| ⟨*point*⟩ `=|=` ⟨*point*⟩
| ⟨*point*⟩ ‘`<`’ ⟨*point*⟩
| ⟨*point*⟩ ‘`>`’ ⟨*point*⟩
| ⟨*point*⟩ `<=` ⟨*point*⟩
| ⟨*point*⟩ `>=` ⟨*point*⟩
| ⟨*string*⟩ `==` ⟨*string*⟩
| ⟨*string*⟩ `=|=` ⟨*string*⟩
| ⟨*string*⟩ ‘`<`’ ⟨*string*⟩
| ⟨*string*⟩ ‘`>`’ ⟨*string*⟩
| ⟨*string*⟩ `<=` ⟨*string*⟩
| ⟨*string*⟩ `>=` ⟨*string*⟩
| ⟨*table*⟩ `==` ⟨*table*⟩
| ⟨*table*⟩ `=|=` ⟨*table*⟩

⟨*textbol*⟩ ::= ‘`(`’ ⟨*boolean*⟩ ‘`)`’
| ⟨*bltinbol*⟩
| ⟨*basicbol*⟩
| ⟨*mixtbol*⟩
| ⟨*plugins_bol*⟩

⟨*boolean*⟩ ::= ⟨*textbol*⟩ | ‘`?`’ ⟨*bolconv*⟩

⟨*bolconv*⟩ ::= ⟨*bolconvar*⟩ ⟨*action*⟩

⟨*bolconvar*⟩ ::= ⟨*bolblkvar*⟩ | ⟨*bolxxxvar*⟩

⟨*bolblkvar*⟩ ::= ‘`(`’ ⟨*rvalue*⟩ ‘`)`’

⟨*bolxxxvar*⟩ ::= ⟨*xxxvar*⟩

⟨*bolvar*⟩ ::= ⟨*boolean*⟩ | ⟨*blkvar*⟩

Tabele

⟨*tablist*⟩ ::= ⟨*tabelist*⟩ | tablist ‘`,`’ ⟨*tabelist*⟩

⟨*tabelist*⟩ ::= ⟨*rvalue*⟩

⟨*tabfrm*⟩ ::= ‘`{`’ ⟨*tablist*⟩ ‘`}`’

⟨*bltintab*⟩ ::= ‘!’ ⟨*tabfrm*⟩
| ⟨*table*⟩ ‘{’ rangelist ‘}’
| `ARGS` | `ARGS` ‘(’ ⟨*at*⟩ ‘)’
| `VAR` | `VAR` ‘(’ ⟨*at*⟩ ‘)’

⟨*basictab*⟩ ::= ⟨*table*⟩ ‘+’ ⟨*table*⟩ | ⟨*table*⟩ ‘*’ ⟨*number*⟩

⟨*mixttab*⟩ ::= ⟨*string*⟩ ‘/’ ⟨*string*⟩
| ⟨*number*⟩ ‘+’ ⟨*table*⟩
| ⟨*number*⟩ ‘*’ ⟨*table*⟩
| ⟨*number*⟩ ‘-’ ⟨*table*⟩
| ⟨*number*⟩ ‘/’ ⟨*table*⟩
| ⟨*number*⟩ ‘%’ ⟨*table*⟩
| ⟨*number*⟩ ‘&’ ⟨*table*⟩
| ⟨*number*⟩ ‘|’ ⟨*table*⟩
| ⟨*number*⟩ ‘^’ ⟨*table*⟩
| ⟨*table*⟩ ‘+’ ⟨*number*⟩
| ⟨*table*⟩ ‘-’ ⟨*number*⟩
| ⟨*table*⟩ ‘/’ ⟨*number*⟩
| ⟨*table*⟩ ‘%’ ⟨*number*⟩
| ⟨*table*⟩ ‘&’ ⟨*number*⟩
| ⟨*table*⟩ ‘|’ ⟨*number*⟩
| ⟨*table*⟩ ‘^’ ⟨*number*⟩
| ⟨*point*⟩ ‘+’ ⟨*table*⟩
| ⟨*point*⟩ ‘-’ ⟨*table*⟩
| ⟨*point*⟩ ‘*’ ⟨*table*⟩
| ⟨*table*⟩ ‘-’ ⟨*point*⟩
| ⟨*string*⟩ ‘+’ ⟨*table*⟩
| ⟨*string*⟩ ‘-’ ⟨*table*⟩
| ⟨*string*⟩ ‘%’ ⟨*table*⟩
| ⟨*table*⟩ ‘+’ ⟨*string*⟩
| ⟨*table*⟩ ‘-’ ⟨*string*⟩
| ⟨*table*⟩ ‘%’ ⟨*string*⟩
| ⟨*boolean*⟩ `&&` ⟨*table*⟩
| ⟨*boolean*⟩ `||` ⟨*table*⟩
| ⟨*boolean*⟩ `^^` ⟨*table*⟩
| ⟨*table*⟩ `&&` ⟨*boolean*⟩
| ⟨*table*⟩ `||` ⟨*boolean*⟩
| ⟨*table*⟩ `^^` ⟨*boolean*⟩

⟨*opertab*⟩ ::= ⟨*fncoper*⟩ ‘(’ ⟨*table*⟩ ‘,’ ⟨*rvalue*⟩ ‘)’ ⟨*iterate*⟩
| ⟨*fncoper*⟩ ‘(’ ⟨*blkvar*⟩ ‘,’ ⟨*table*⟩ ‘)’ ⟨*iterate*⟩
| ⟨*fncoper*⟩ ‘(’ ⟨*tabvar*⟩ ‘)’ ⟨*iterate*⟩
| ⟨*ppmmoper*⟩ ‘(’ ⟨*tabvar*⟩ ‘)’ ⟨*iterate*⟩

⟨*texttab*⟩ ::= ‘(’ ⟨*table*⟩ ‘)’
| ⟨*basictab*⟩
| ⟨*bltintab*⟩
| ⟨*mixttab*⟩

| ‘!’ ⟨opertab⟩
| ⟨plugins_tab⟩

⟨table⟩ ::= ⟨texttab⟩
| ‘!’ ⟨tabconv⟩

⟨tabconv⟩ ::= ⟨tabconvar⟩ ⟨action⟩

⟨tabconvar⟩ ::= ⟨tabblkvar⟩ | ⟨tabxxxvar⟩

⟨tabblkvar⟩ ::= ‘(’ ⟨rvalue⟩ ‘)’

⟨tabxxxvar⟩ ::= ⟨xxxvar⟩

⟨tabvar⟩ ::= ⟨table⟩ | ⟨blkvar⟩

Hashes

⟨hshlist⟩ ::= ⟨hshelist⟩ | ⟨hshlist⟩ ‘,’ ⟨hshelist⟩

⟨hshelist⟩ ::= ⟨strvar⟩ ‘:’ ⟨rvalue⟩

⟨hshfrm⟩ ::= ‘[’ ⟨hshlist⟩ ‘]’

⟨bltinhsh⟩ ::= ‘#’ ⟨hshfrm⟩

⟨basichsh⟩ ::= ⟨hash⟩ ‘+’ ⟨hash⟩
| ⟨hash⟩ ‘|’ ⟨hash⟩
| ⟨hash⟩ ‘*’ ⟨hash⟩
| ⟨hash⟩ ‘&’ ⟨hash⟩
| ⟨hash⟩ ‘-’ ⟨hash⟩
| ⟨hash⟩ ‘/’ ⟨hash⟩

⟨texthsh⟩ ::= ‘(’ ⟨hash⟩ ‘)’
| ⟨basichsh⟩
| ⟨bltinhsh⟩
| ⟨plugins_hsh⟩

⟨hash⟩ ::= ⟨texthsh⟩
| ‘#’ ⟨hshconv⟩

⟨hshconv⟩ ::= ⟨hshconvar⟩ ⟨action⟩

⟨hshconvar⟩ ::= ⟨hshblkvar⟩ | ⟨hshxxxvar⟩

⟨hshblkvar⟩ ::= ‘(’ ⟨rvalue⟩ ‘)’

⟨hshxxxvar⟩ ::= ⟨xxxvar⟩

⟨hshvar⟩ ::= ⟨hash⟩ | ⟨blkvar⟩

6.2.2 Sintaxa adaugata

Extensie intrare/iesire

⟨*io_wait*⟩ ::= `wait` ⟨*strvar*⟩ ‘;’

⟨*io_format*⟩ ::= `format nul` ‘=’ ⟨*strvar*⟩ ‘;’
| `format integer` ‘=’ ⟨*strvar*⟩ ‘,’ ⟨*bolvar*⟩ ‘;’
| `format double` ‘=’ ⟨*strvar*⟩ ‘;’
| `format point` ‘=’ ⟨*strvar*⟩ ‘,’ ⟨*strvar*⟩ ‘,’ ⟨*strvar*⟩ ‘;’
| `format string` ‘=’ ⟨*strvar*⟩ ‘,’ ⟨*strvar*⟩ ‘;’
| `format boolean` ‘=’ ⟨*strvar*⟩ ‘,’ ⟨*strvar*⟩ ‘;’
| `format table` ‘=’ ⟨*strvar*⟩ ‘,’ ⟨*strvar*⟩ ‘,’ ⟨*strvar*⟩ ‘;’
| `format hash` ‘=’ ⟨*strvar*⟩ ‘,’ ⟨*strvar*⟩ ‘,’ ⟨*strvar*⟩ ‘,’ ⟨*strvar*⟩ ‘,’ ⟨*strvar*⟩ ‘,’ ⟨*strvar*⟩ ‘;’

arrays

⟨*io_formatab*⟩ ::= `format` ⟨*valtype*⟩

Pluginul necesar “bltin”

data

⟨*bltin_ret*⟩ ::= ⟨*bltin_rvalues*⟩ ⟨*at*⟩ ‘;’

⟨*bltin_new*⟩ ::= `new` ⟨*bltin_new1*⟩

⟨*bltin_new1*⟩ ::= ⟨*newtype*⟩ ⟨*bltin_newlist*⟩ ⟨*at*⟩ ‘;’

⟨*newtype*⟩ ::= ⟨*valtype*⟩ | ⟨*empty*⟩

⟨*bltin_newlist*⟩ ::= ⟨*newididxs*⟩ | ⟨*bltin_newlist*⟩ ‘,’ ⟨*newididxs*⟩

rvalues

⟨*bltin_rvalues*⟩ ::= `read` ⟨*valtype*⟩ | ⟨*number*⟩ | ⟨*point*⟩ | ⟨*string*⟩ | ⟨*boolean*⟩ | ⟨*table*⟩ | ⟨*hash*⟩

Variabile statice

data

⟨*static*⟩ ::= `static` ⟨*static_list*⟩

⟨*static_list*⟩ ::= ⟨*static_item*⟩ | ⟨*static_list*⟩ ’,’ ⟨*static_item*⟩

⟨*static_item*⟩ ::= `IDENTIFIER`

Functii matematice

numbers

$\langle numrem \rangle$ ::= `rem` '(' $\langle numvar \rangle$ '`,`' $\langle numvar \rangle$ ')'

$\langle math \rangle$ ::= $\langle math1 \rangle$ $\langle pntdbl \rangle$ ::= $\langle pntdbl1 \rangle$

$\langle math1 \rangle$ ::= `abs` '(' $\langle numvar \rangle$ ')'
| `sgn` '(' $\langle numvar \rangle$ ')'
| `floor` '(' $\langle numvar \rangle$ ')'
| `ceil` '(' $\langle numvar \rangle$ ')'
| `sin` '(' $\langle numvar \rangle$ ')'
| `cos` '(' $\langle numvar \rangle$ ')'
| `tan` '(' $\langle numvar \rangle$ ')'
| `asin` '(' $\langle numvar \rangle$ ')'
| `acos` '(' $\langle numvar \rangle$ ')'
| `atan` '(' $\langle numvar \rangle$ ')'
| `sec` '(' $\langle numvar \rangle$ ')'
| `csc` '(' $\langle numvar \rangle$ ')'
| `cot` '(' $\langle numvar \rangle$ ')'
| `r2d` '(' $\langle numvar \rangle$ ')'
| `d2r` '(' $\langle numvar \rangle$ ')'
| `exp` '(' $\langle numvar \rangle$ ')'
| `log` '(' $\langle numvar \rangle$ ')'
| `exp2` '(' $\langle numvar \rangle$ ')'
| `log2` '(' $\langle numvar \rangle$ ')'
| `exp10` '(' $\langle numvar \rangle$ ')'
| `log10` '(' $\langle numvar \rangle$ ')'
| `sqrt` '(' $\langle numvar \rangle$ ')'
| `sinh` '(' $\langle numvar \rangle$ ')'
| `cosh` '(' $\langle numvar \rangle$ ')'
| `tanh` '(' $\langle numvar \rangle$ ')'
| `asinh` '(' $\langle numvar \rangle$ ')'
| `acosh` '(' $\langle numvar \rangle$ ')'
| `atanh` '(' $\langle numvar \rangle$ ')'

$\langle pntdbl1 \rangle$ ::= `mod` '(' $\langle pntvar \rangle$ ')' | `ang` '(' $\langle pntvar \rangle$ ')'

points

$\langle pntdiv \rangle$ ::= $\langle point \rangle$ '`/`' $\langle point \rangle$

Rendare de text bazata pe biblioteca "freetype"

primitives

$\langle ft\_wff \rangle$ ::= `font` '(' $\langle string \rangle$ ')'
| `face` '(' $\langle string \rangle$ ')'
| `face` '(' $\langle number \rangle$ ')'
| `font` '(' $\langle number \rangle$ ')'
| `font` '(' $\langle pntarg \rangle$ ')'

$\quad | \;$ `align` '(' $\langle strvar \rangle$ ')'
$\quad | \;$ `write` '(' $\langle strvar \rangle$ ',' $\langle numvar \rangle$ ',' $\langle pntarg \rangle$ ')'
$\quad | \;$ `write` '(' $\langle strvar \rangle$ ',' $\langle numvar \rangle$ ')'
$\quad | \;$ `write` '(' $\langle strvar \rangle$ ')'

arrays

$\langle ft\_font \rangle$ ::= `font` '(' ')'

Zpoints

$\langle ft\_bbox \rangle$ ::= `bbox` '(' $\langle strvar \rangle$ ',' $\langle numvar \rangle$ ',' $\langle pntarg \rangle$ ')'
$\quad | \;$ `bbox` '(' $\langle strvar \rangle$ ',' $\langle numvar \rangle$ ')'
$\quad | \;$ `bbox` '(' $\langle strvar \rangle$ ')'

Interfata sistem

flow

$\langle shell \rangle$ ::= `shell` $\langle strvar \rangle$ ';'

numbers

$\langle sys\_num \rangle$ ::= `pid` | `ppid`

strings

$\langle sys\_str \rangle$ ::= `user` | `host`

arrays

$\langle sys\_tab \rangle$ ::= `uname` | `time`

Puncte la distanta minima de 1 cm/mm/in/pt

primitives

$\langle unit\_use \rangle$ ::= `unit` '(' number ',' string ')' | `unit` '(' point ')'

Blocul "if-else"

flow

$\langle ifelse \rangle$ ::= $\langle if \rangle$ | $\langle if \rangle$ `else` $\langle block \rangle$ | $\langle if \rangle$ `else` $\langle ifelse \rangle$

$\langle if \rangle$ ::= `if` '(' $\langle bolvar \rangle$ ')' $\langle block \rangle$

Bloc "case"

flow

```
⟨case⟩         ::= case ” ⟨cases⟩ ”

⟨cases⟩        ::= ⟨casee_num⟩ ⟨cases_num⟩
                 | ⟨casee_str⟩ ⟨cases_str⟩
                 | ⟨casee_pnt⟩ ⟨cases_pnt⟩

⟨casee_num⟩ ::= ‘(’ ⟨number⟩ ‘)’

⟨cases_num⟩ ::= ⟨empty⟩
              | ⟨case_num_list⟩ ‘;’ ⟨block⟩ ⟨cases_num⟩
              | else ⟨block⟩

⟨case_num_list⟩ ::= ⟨case_num_item⟩ | ⟨case_num_list⟩ ‘:’ ⟨case_num_item⟩

⟨case_num_item⟩ ::= ⟨numvar⟩ | ⟨numvar⟩ ‘,’ ⟨numvar⟩

⟨casee_str⟩   ::= ‘(’ ⟨string⟩ ‘)’

⟨cases_str⟩   ::= ⟨empty⟩
                | ⟨case_str_list⟩ ‘;’ ⟨block⟩ ⟨cases_str⟩
                | else ⟨block⟩

⟨case_str_list⟩ ::= ⟨case_str_item⟩ | ⟨case_str_list⟩ ‘:’ ⟨case_str_item⟩

⟨case_str_item⟩ ::= ⟨string⟩

⟨casee_pnt⟩   ::= ‘(’ ⟨point⟩ ‘)’

⟨cases_pnt⟩   ::= ⟨empty⟩
                | ⟨case_pnt_list⟩ ‘;’ ⟨block⟩ ⟨cases_pnt⟩
                | else ⟨block⟩

⟨case_pnt_list⟩ ::= ⟨case_pnt_item⟩ | ⟨case_pnt_list⟩ ‘:’ ⟨case_pnt_item⟩

⟨case_pnt_item⟩ ::= ⟨point⟩
```

Corespondenta intre operatori literali si reprezentarea “type” a lor ca string

strings

```
⟨oper_str⟩ ::= operator ⟨asgoper⟩
           | operator ⟨rmoper⟩
           | operator ⟨metaoper⟩
           | ‘<’ ‘>’ operator
           | operator ⟨ppmmoper⟩
           | ⟨ppmmoper⟩ operator
           | operator ‘.’ ⟨fncoper⟩
           | operator ⟨fncoper⟩
           | operator ⟨cmpoper⟩
           | operator ⟨cnvoper⟩
           | operator ‘=’
           | operator ‘‘’
```

Programare OO bazata pe paradigma "class"

data

⟨class_new⟩ ::= class ⟨class_name⟩ ⟨class_inherit⟩ '{' ⟨class_operators⟩ ⟨class_methods⟩
⟨class_datas⟩ ⟨class_friends⟩ '}'

⟨class_name⟩ ::= ⟨identifier⟩

⟨class_inherit⟩ ::= ⟨empty⟩ | ⟨class_item⟩

⟨class_item⟩ ::= ':' ⟨identifier⟩

⟨class_operators⟩ ::= ⟨empty⟩ | ⟨class_operator⟩ ';' ⟨class_operators⟩

⟨class_operator⟩ ::= ⟨class_op_left⟩ operator ⟨class_item⟩
| operator ⟨class_op_right⟩ ⟨class_item⟩

⟨class_op_right⟩ ::= ⟨asgoper⟩
| ⟨fncoper⟩
| ⟨cmpoper⟩
| ‘‘’
| '='
| ⟨rmoper⟩
| ⟨ppmmoper⟩
| operator
| '(' ')'

⟨class_op_left⟩ ::= ⟨fncoper⟩
| ⟨ppmmoper⟩
| ⟨cnvoper⟩
| ⟨metaoper⟩

⟨class_methods⟩ ::= ⟨empty⟩ | ⟨class_method⟩ ';' ⟨class_methods⟩

⟨class_method⟩ ::= method ⟨identifier⟩ ⟨class_item⟩

⟨class_datas⟩ ::= ⟨empty⟩ | data ⟨class_datas1⟩ ';'

⟨class_datas1⟩ ::= ⟨class_data⟩ | ⟨class_datas1⟩ ',' ⟨class_data⟩

⟨class_data⟩ ::= ⟨class_data1⟩ ⟨class_data_init⟩

⟨class_data1⟩ ::= ⟨identifier⟩ | const ⟨identifier⟩

⟨class_data_init⟩ ::= ⟨empty⟩ | '=' ⟨rvalue⟩

⟨class_friends⟩ ::= ⟨empty⟩ | friend ⟨class_friends1⟩ ';'

⟨class_friends1⟩ ::= ⟨class_friend⟩ | ⟨class_friends1⟩ ',' ⟨class_friend⟩

⟨class_friend⟩ ::= ⟨type⟩

identifiers

⟨*class_base_id*⟩ ::= `base`

strings

⟨*class_class*⟩ ::= `class` ‘`(`’ ⟨*blkvar*⟩ ‘`)`’

hook

⟨*class_hook*⟩ ::= ‘`:`’ ⟨*class_base_self*⟩ ‘`(`’ ⟨*tablist*⟩ ‘`)`’

⟨*class_base_self*⟩ ::= `base` | `self`

flow

⟨*class_base_new*⟩ ::= `base` ‘`(`’ ⟨*tablist*⟩ ⟨*maketab*⟩ ‘`)`’

⟨*class_self_new*⟩ ::= `self` ‘`(`’ ⟨*tablist*⟩ ⟨*maketab*⟩ ‘`)`’

Bucla cu un numar fix de repetari: “loop”

flow

⟨*loop*⟩ ::= `loop` ‘`(`’ ⟨*numvar*⟩ ‘`)`’ ⟨*block*⟩

Bucla cu intrare conditionata: “while”

flow

⟨*while*⟩ ::= `while` ‘`(`’ ⟨*blackhole*⟩ ‘`)`’ ⟨*block*⟩

Iteratia “for”

flow

⟨*for*⟩ ::= `for` ‘`(`’ ⟨*blackhole*⟩ ‘`:`’ ⟨*blackhole*⟩ ‘`:`’ ⟨*blackhole*⟩ ‘`)`’ ⟨*block*⟩

Iteratatii prin elementele unui tablou sau colectie: “foreach”

flow

⟨*foreach_tab*⟩ ::= `foreach` ‘`(`’ ⟨*table*⟩ ‘`)`’ ⟨*block*⟩

⟨*foreach_hsh*⟩ ::= `foreach` ‘`(`’ ⟨*hash*⟩ ‘`)`’ ⟨*block*⟩

6.3 Precedenta operatorilor

Tip	Aritate	Operator	Descriere	Asociativitate	Prec.	Exemplu
Atribuire	binar	=	atribuire	fara	1	a, b = c
Atribuire	binar	+=	+atribuire	fara	1	a += 10
Atribuire	binar	-=	-atribuire	fara	1	a -= b
Atribuire	binar	*=	*atribuire	fara	1	a, b *= c
Atribuire	binar	/=	/atribuire	fara	1	a /= b
Atribuire	binar	%=	%atribuire	fara	1	a %= b
Atribuire	binar	=	atribuire	fara	1	a = b
Atribuire	binar	&=	&atribuire	fara	1	a, b &= c
Atribuire	binar	$\vert=$	$\vert$ *atribuire*	fara	1	a $\vert= b$
Atribuire	binar	$\sim=$	$\sim$ *atribuire*	fara	1	a, b $\sim= c$
Relational	binar	==	egalitate	stanga	2	a == b
Relational	binar	$=\vert=$	inegalitate	stanga	2	a $=\vert= b$
Relational	binar	$<$	mai mic	stanga	3	a $<$ b
Relational	binar	$>$	mai mare	stanga	3	a $>$ b
Relational	binar	$\leq$	mai mic sau egal	stanga	3	a $\leq b$
Relational	binar	$\geq$	mai mare sau egal	stanga	3	a $\geq b$
Logic	binar	$\uparrow\uparrow$	sau-exclusiv	stanga	4	a $\uparrow\uparrow b$
Logic	binar	$\vert\vert$	sau	stanga	5	a $\vert\vert\, b$
Logic	binar	&&	si	stanga	6	a && b
Logic	unar	$\sim\sim$	negare	dreapta	7	a == $\sim\sim b$
Aritmetic	binar	+	adunare	stanga	8	a + b
Aritmetic	binar	-	scadere	stanga	9	a - b
Aritmetic	binar	*	inmultire	dreapta	10	a * b
Aritmetic	binar	/	impartire	dreapta	11	a / b
Aritmetic	binar	%	modulo	stanga	12	a % b
Aritmetic	binar	$\vert$	sau	stanga	13	a $\vert\, b$
Aritmetic	binar	&	si	stanga	14	a & b
Aritmetic	binar	$\uparrow$	exponent	dreapta	15	a $\uparrow$ b
Aritmetic	binar	$\sim$	concatentare	stanga	16	a $\sim b$

Tip	Aritate	Operator	Descriere	Asociativitate	Prec.	Exemplu
Aritmetic	unar	+	plus	stanga	17	+a
Aritmetic	unar	-	minus	stanga	17	-a
Aritmetic	unar	*	asterisc	dreapta	17	* a
Aritmetic	unar	/	slash	dreapta	17	/a
Aritmetic	unar	%	procent	stanga	17	% a
Aritmetic	unar	$\vert$	bara	stanga	17	$\vert$ a
Aritmetic	unar	&	ampersand	stanga	17	& a
Aritmetic	unar	$\uparrow$	circumflex	dreapta	17	$\uparrow$ a
Aritmetic	unar	$\sim$	tilda	stanga	17	$\sim$ a
Incrementare	unar	++	post	fara	18	a++
Decrementare	unar	−−	post	fara	18	a−−
Incrementare	unar	++	pre	fara	19	++a
Decrementare	unar	−−	pre	fara	19	−−a
Conversie	unar	$	numar	fara	20	$ n
Conversie	unar	@	punct	fara	20	@p
Conversie	unar	"	string	fara	20	" s
Conversie	unar	?	boolean	fara	20	?b
Conversie	unar	!	tablou	fara	20	!t
Conversie	unar	#	hash	fara	20	# h
Selectie	binar	{}	reuniune intervale	stanga	21	tab{1,2:,}
Apel	unar	‘()	apel metoda anonima	stanga	22	self‘()
Apel	unar	‘	apel metoda	stanga	23	a‘b (c)
Conversie	unar	<>	conversie la tip	dreapta	24	<obj>self
Indexare	unar	[] sau .	index, camp, cheie	stanga	1	a[1].b[’c’]

6.4 Iesirea gramaticii exemplu

```
State 1 contains 1 shift/reduce conflict.

Grammar
rule 1    x -> i j
rule 2    x -> k l
rule 3    i -> 'a'
rule 4    j -> 'b' 'a'
rule 5    k -> 'a' 'b'
rule 6    l -> 'a'

Terminals, with rules where they appear

$ (-1)
'a' (97) 3 4 5 6
'b' (98) 4 5
error (256)

Nonterminals, with rules where they appear
```

```
x (5)
    on left: 1 2
i (6)
    on left: 3, on right: 1
j (7)
    on left: 4, on right: 1
k (8)
    on left: 5, on right: 2
l (9)
    on left: 6, on right: 2


state 0

    'a'         shift, and go to state 1

    x           go to state 10
    i           go to state 2
    k           go to state 3



state 1

    i  ->  'a' .   (rule 3)
    k  ->  'a' . 'b'   (rule 5)

    'b'         shift, and go to state 4

    'b'         [reduce using rule 3 (i)]
    $default    reduce using rule 3 (i)



state 2

    x  ->  i . j   (rule 1)

    'b'         shift, and go to state 5

    j           go to state 6



state 3

    x  ->  k . l   (rule 2)

    'a'         shift, and go to state 7
```

```
    l           go to state 8



state 4

    k  ->  'a' 'b' .   (rule 5)

    $default    reduce using rule 5 (k)



state 5

    j  ->  'b' . 'a'   (rule 4)

    'a'         shift, and go to state 9



state 6

    x  ->  i j .   (rule 1)

    $default    reduce using rule 1 (x)



state 7

    l  ->  'a' .   (rule 6)

    $default    reduce using rule 6 (l)



state 8

    x  ->  k l .   (rule 2)

    $default    reduce using rule 2 (x)



state 9

    j  ->  'b' 'a' .   (rule 4)
```

```
    $default    reduce using rule 4 (j)


state 10

    $           go to state 11


state 11

    $           go to state 12


state 12

    $default    accept
```