Universitat de Girona
Escola Politècnica Superior

Projecte/Treball Final de Carrera

Estudi: Eng. Tèc. Informàtica de Sistemes

Títol:

Traductor de pseudocodi a java

Document: Memòria

Alumne: Carles Royán Salvatella

Director/Tutor: Josep Suy
Departament: Informàtica i Matemàtica Aplicada
Àrea: Llenguatges i Sistemes Informàtics

Convocatòria: Juny 2003

“Programs that write programs are the
happiest programs in the world.”
(Andrew Hume)

Índex

1. Introducció.....6
1.1 Objectius principals.....7
1.2 Altres objectius.....7
1.3 Introducció al llenguatge a compilar.....8
1.4 Convencions.....9

2. Què és un compilador?.....11
2.1 Definició.....12
2.2 Estructura d'un compilador.....13
2.3 Anàlisi lèxica.....15
2.4 Anàlisi sintàctica.....17
2.4.1 Gramàtiques lliures de context.....17
2.4.2 Classes d'analitzadors sintàctics.....18
2.5 Anàlisi semàntica.....19
2.6 Administració de la taula de símbols.....20
2.7 Tractament d'errors.....20
2.8 Codi intermedi, optimització i generació de codi final.....21
2.9 Compilació dirigida per la sintaxi.....22
2.10 Marc de treball del traductor implementat.....23

3. Pseudocodi 03.....24
3.1 Introducció al pseudocodi.....25
3.2 Conceptes bàsics.....25
3.2.1 Comentaris.....26
3.2.2 Sentències.....27
3.2.3 Blocs de codi.....27
3.2.4 Paraules reservades.....27
3.2.5 Identificadors.....28
3.2.6 Literals i tipus de dades.....28
3.2.7 Expressions i operadors.....29
3.3 Tipus de dades: declaració de variables i instanciació.....30
3.4 Instruccions acceptades.....32
3.4.1 Instrucció d'assignació.....32
3.4.2 Instrucció res.....32
3.4.3 Instruccions alternatives: si i cas.....33
3.4.4 Instruccions iteratives: mentre i per.....34
3.4.5 Instrucció crear.....35
3.4.6 Instrucció destruir.....36
3.4.7 Instrucció retorna.....36
3.4.8 Instruccions d'E/S.....36
3.5 Esquema d'algorisme.....38
3.6 Classes.....39
3.6.1 Encapsulament.....40
3.6.2 Atributs i mètodes.....41
3.6.3 Atributs de classe i mètodes de classe.....41
3.6.4 Atributs i funcions privades.....43
3.6.5 Constructors i destructors.....44

3.6.6 Sobrecàrrega....45
3.6.7 Herència....45
3.6.8 Les referències jo i super....46
3.7 Taules....47
3.8 Ús de les classes....48
3.9 Format del fitxer font....49
3.10 Restriccions del traductor....50

4. Entorn de treball....51
4.1 Eines per la generació de compiladors....52
4.2 Diferents alternatives....52
4.2.1 Generadors ascendents....53
4.2.1.1 LEX i YACC....53
4.2.1.2 FLEX i BISON....54
4.2.2 Generadors descendents....54
4.2.2.1 PCCTS 1.33....55
4.2.2.2 ANTLR 2.71....55
4.2.3 Conclusió....56
4.3 Instal·lació de l'entorn de treball....57

5. Construcció del traductor....58
5.1 Anàlisi lèxica: classe CpcLexer....59
5.1.1 Paraules reservades....59
5.1.2 Identificadors....60
5.1.3 Literals....60
5.1.4 Separadors i operadors....61
5.1.5 Tokens a ignorar....62
5.2 Anàlisi sintàctica: classe CpcParser....62
5.3 Anàlisi semàntica: comprovacions i missatges....66
5.3.1 Errors en la declaració de variables....66
5.3.2 Errors en la declaració de constants....66
5.3.3 Errors en la declaració de mètodes/funcions....67
5.3.4 Errors en les sentències....67
5.3.5 Errors en les expressions....69
5.3.6 Errors en l'accés a les classes....69
5.4 Tractament d'errors....71
5.5 La taula de símbols....74
5.5.1 Estructura de dades....74
5.5.2 Tractament dels àmbits....75
5.5.3 Informació desada....77
5.6 Classes auxiliars: ParamInfo i FuncInfo....77
5.7 Generació de codi: classe CpcGenJava....78

6. Proves d'execució....80
6.1 Les proves....81
6.2 Exemple 1: números primers....81
6.2 Exemple 2: classe Punt3d....83
6.3 Exemple 3: llibreria matemàtica....86

7. Aspectes finals.....94
7.1 Temporització orientativa.....95
7.2 Problemes.....96
7.3 Treballs futurs.....97
7.4 Conclusió.....98
7.5 Bibliografia.....99
7.6 Altres recursos d'interès.....99
7.7 Agraïments.....100

1. Introducció

1.1 Objectius principals

En les diferents assignatures de programació de les carreres informàtiques, els professors donen als alumnes diversos codis font de programes d'exemple o d'estructures de dades. Una eina que pugui permetre la compilació i execució de pseudocodi pot ésser molt útil. Per exemple, a l'alumnat els hi permet concentrar-se en el propi algorisme i deixar de banda la rigidesa de qualsevol llenguatge de programació. També els ajuda a preparar les proves d'avaluació així com entendre millor el funcionament de programes o estructures de dades.

L'objectiu del projecte és emplenar aquest buit amb una eina capaç de traduir el pseudocodi al llenguatge de programació **Java**, utilitzant la programació orientada a objectes tant en la realització del projecte, així com en el pseudocodi a tractar. D'aquesta manera s'obtindrà un programa en **Java** del qual es podrà comprovar el seu bon funcionament. El projecte estarà basat em teories de llenguatges i creació d'autòmats reconeixedors de gramàtiques, ja que són els fonaments per tal de realitzar un compilador/traductor.

El traductor rebrà com a entrada un fitxer en format text del programa, escrit en llenguatge pseudocodi, que es desitja traduir. Al finalitzar aquest procés, si hi ha hagut errors els mostrarà per pantalla, i si no n'hi ha hagut cap, obtindrem el fitxer **Java** traduït. Aquest fitxer podrà ésser compilat per qualsevol compilador estàndard de **Java**, com el subministrat per *Sun Microsystems*.

En el traductor s'implementarà tant l'anàlisi lèxica, com la sintàctica i la semàntica, que són les parts principals de les que es composa un compilador. La generació de codi intermedi i la seva corresponent optimització, seran reemplaçades per la generació de codi **Java**.

1.2 Altres objectius

El projecte conté altres objectius com ara conèixer les fases d'un compilador, ja que existeix poca diferència entre un traductor a un llenguatge de programació i un compilador. En la construcció del traductor s'intentarà modularitzar la seva implementació en cada una de les fases en que es divideix conceptualment un compilador.

També hi ha l'objectiu d'aplicar els coneixements adquirits en moltes assignatures de la carrera informàtica, entre d'altres, "*estructures de dades i algorismes*", "*llenguatge, gramàtiques i autòmats*", "*compiladors*", ...

Finalment, es troba com a objectius aprendre a extreure la informació necessària de llibres i manuals molt extens i a expressar de manera escrita, organitzada i estructurada (la memòria) tot el treball realitzat.

1.3 Introducció al llenguatge a compilar

A continuació es remarcaran les característiques principals del pseudocodi a traduir. En el capítol 3 s'explica amb molt més detall el llenguatge.

El pseudocodi tractat és diferent al que hi havia en anys anteriors. Els llenguatges orientats a objectes s'han guanyat la preferència dels programadors, i són el present de la informàtica, en detriment dels llenguatges de programació estructurada. Calia que el pseudocodi s'adaptés a aquests llenguatges, fent-lo fortament orientat a objectes i proporcionant eines per tal poder abordar aquest model de programació. Aquí va sorgir el pseudocodi 01, el qual tenia com a base el pseudocodi habitual amb algunes modificacions i afegits, per tal d'explotar el model d'orientació a objectes. En aquest projecte s'implementa el pseudocodi 03, que parteix del pseudocodi 01, amb algunes modificacions per tal de fer-lo més pràctic en el seu ús real.

Existeixen els tipus elementals: enter, real, caràcter, booleà; i el tipus referència. Aquest últim el forma variables declarades com a qualsevol tipus definit per l'usuari en una classe. Aquestes variables referencien la instància de la classe (que ha de ser creada posteriorment).

Existeixen les instruccions repetitives, condicionals i l'assignació, d'igual manera que en el pseudocodi habitual.

Es disposa de rutines predefinides d'entrada i sortida:

- *escriure(expressió)*; escriu per la sortida estàndard (normalment la pantalla) l'expressió o variable, independent del tipus.

- *variable<-llegir();* llegeix de l'entrada estàndard el valor d'una variable de tipus elemental i retorna el valor a la retorna.

Les taules són tractades com objectes, tot i que els seus elements són accedits de la manera tradicional (utilitzant: *taula1[enter]*). A més, tenen definit un mètode per tal de saber la seva llargària:

- *longitud()*; retorna un enter amb la llargària total amb què s'ha instanciat la taula.

El tipus cadena no existeix com a tipus elemental, però es pot implementar com a una classe. En canvi, si existeix el literal cadena, implementat a l'estil **C/C++** o **Java**, és a dir, una seqüència de caràcters tancada entre cometes dobles (“).

Finalment, cal dir que el pseudocodi no és sensible a majúscules i minúscules (no és *case sensitive*). Això significa que una variable tota en majúscules i una altra variable tota en minúscules són interpretades com la mateixa variable. Això fa més fàcil la programació per els novells, que solen patir molt per la característica de distincions de majúscules i minúscules del **C++** i del **Java**.

1.4 Convencions

A continuació, s'explicaran diverses convencions utilitzades al llarg de la memòria, per tal que sigui més uniforme i no calgui explicar-los contínuament.

Per a mostrar elements que siguin opcionals, es farà entre corxets, d'aquesta manera:

si condició **llavors** instrucció [**altrament** instrucció] **fsi**

Denota que “**altrament** instrucció” és opcional, i per tant, es pot ometre.

Quan es faci referència a algun llenguatge de programació, s'escriurà el nom d'aquest en negreta, mentre que el nom d'empreses vinculades a algun llenguatge o compilador, s'escriurà en lletra cursiva. També s'escriurà en cursiva paraules tècniques i anglicismes acceptats en el món dels compiladors. Cal esmentar que els noms de fitxers aniran entre cometes, així com signes estranys, per tal que no es confonguin amb el text.

Finalment, a la pàgina següent hi ha la taula 1.1 en la que es pot observar el conveni per a la nomenclatura d'identificadors, el qual és seguit en aquesta memòria, així com en el codi del projecte.

Tipus d'identificador	Conveni	Exemples
Noms de classes	Cada paraula dintre l'identificador comença per majúscula.	Natural, ClasseFraccio
Noms de funcions	Cada paraula dintre de l'identificador comença per majúscula, excepte la primera.	esborrar, esborrarClau
Noms de variables	Cada paraula dintre de l'identificador comença per majúscula, excepte la primera.	clau, clauEntera
Noms de constants	Cada lletra s'escriu en majúscula i el símbol de guió baix s'utilitza per separar cada paraula.	MAX, MAX_REGISTRES

Taula 1.1 Conveni de nomenclatura dels identificadors

2. Què és un compilador?

2.1 Definició

Els ordinadors funcionen executant una sèrie d'instruccions escrites en llenguatge màquina, un llenguatge binari que només entén el computador. Escriure en aquest llenguatge és una feina molt laboriosa, ja que tot es redueix al vocabulari binari, per la qual cosa, pensar en algorismes complexes pot ser una tasca gairebé impossible. Per facilitar la programació, va sorgir el llenguatge assemblador, el qual no dista gaire del codi màquina, no obstant les instruccions equivalen a abreviatures de tres lletres, anomenades mnemotècnics, i les dades que es poden manipular són números en base 10, enlloc de nombres binaris. No obstant, programar d'aquesta forma continua sent molt difícil, ja que problemes resolubles fàcilment a mà poden costar molt d'esforç programar-los en llenguatge assemblador. La necessitat de realitzar programes més complexes va fer sorgir els llenguatges d'alt nivell, en contraposició als llenguatges de baix nivell (codi màquina, assemblador). Aquests llenguatges de programació creen una capa o nivell d'abstracció, la qual cosa fa que el programador pugui obviar les particularitats del maquinari i centrar-se únicament en el problema a resoldre. Exemples de llenguatges d'alt nivell són el **basic**, el **Java** o el **pascal**.

Com hem dit abans, els ordenadors són capaços d'executar codi màquina, per tant, un programa escrit en un llenguatge d'alt nivell no pot ésser executat directament. Els compiladors són programes que tradueixen llenguatges d'alt nivell a codi màquina. Per tant, els compiladors actuen com a traductors.

De compiladors/traductors n'hi ha de molts tipus:

- Compiladors: programes que tradueixen d'un llenguatge d'alt nivell a un llenguatge de baix nivell (generalment codi màquina). Poden generar un fitxer *objecte* el qual ha d'ésser posteriorment enllaçat.
- Intèrprets: programes que tradueixen d'un llenguatge d'alt nivell a codi màquina en temps d'execució.
- Assembladors: tradueixen el llenguatge assemblador a codi màquina.
- Conversos font-font: tradueixen d'un llenguatge d'alt nivell a un altre també d'alt nivell (per exemple de **pascal** a **C** o de **pseudocodi** a **Java,** com el present projecte).

El compilador, a més de realitzar la corresponent traducció, també comprova que el codi escrit en llenguatge d'alt nivell sigui correcte, i en cas de que hi hagin errors (funcions/instruccions mal escrites, etc.) ha d'avisar al usuari per a que aquest els pugui arreglar.

L'existència de compiladors ha facilitat l'expansió de la informàtica de forma que han permès la construcció d'algorismes i programes molt avançats.

2.2 Estructura d'un compilador

A més d'un compilador, pot ésser necessari altres programes per a crear un programa objecte executable. Un programa font, sovint, es divideix en mòduls desats en fitxers diferents. El preprocessador s'encarrega de processar i unir els diferents fitxers abans de començar el procés de compilació. També sol expandir fragments de codi, anomenats macros (molt comú en **C/C++**). Una vegada s'han compilats els diferents fitxers, pot caldre enllaçar els diferents fitxers objecte per tal de crear l'executable binari final. Aquest programa es sol anomenar linkador, el qual enllaça els diferents fitxers objecte ja compilats tot resolent dependències de llibreries de codi o crides a funcions externes.

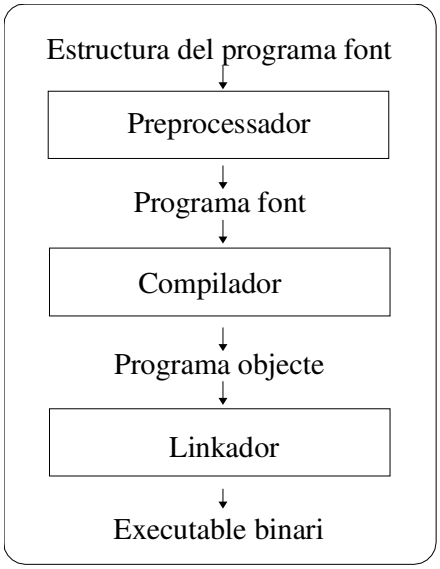


Figura 2.1 Programes auxiliars a un compilador

Conceptualment, un compilador actua per fases, cadascuna de les quals té una representació interna del programa font diferent de les altres. Les fases o etapes estan agrupades en dos parts: *front-end* i *back-end*. Aquesta divisió en dues parts, ve donada per el fet que d'aquesta manera, cal

realitzar un sol *front-end* per llenguatge i un sol *back-end* per arquitectura hardware.

Per exemple, suposem que es vol construir un compilador per els llenguatges **C**, **Pascal** i **APL** i que volem que generin codi per a tres plataformes diferents: *Intel x86*, *Sun Sparc* i *Apple PowerPC*. Per tant, tenim:

- C Intel x86
- Pascal Sun Sparc
- APL Apple PowerPC

Podríem pensar que necessitaríem fer 9 compiladors diferents. Això seria molt costós, ja que s'haurien de fer tres compiladors complets per el mateix llenguatge. Dividim ara la construcció del compilador en *front-end* i *back-end*, sent el primer dependent del llenguatge i el segon depenent del maquinari. D'aquesta manera, hauríem de fer 6 compiladors: tres *front-end*, un per cada llenguatge, i tres *back-end*, un per cada plataforma, i combinar-los.

En la figura 2.2 podem veure com restaria:

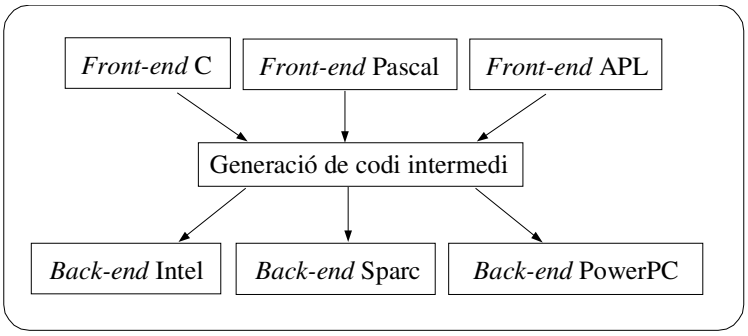


Figura 2.2 Exemple de divisió d'un compilador en *front-end* i *back-end*

Com ja s'ha dit abans, en l'etapa de *front-end* hi ha les fases dependents del llenguatges font a tractar:

- Anàlisi lèxica (*scanner*)
- Anàlisi sintàctica (*parser*)
- Anàlisi semàntica
- Generació de codi intermedi

I el *back-end* el composen les dependents del maquinari:

- Optimitzador
- Generació de codi

Dues activitats, l'administració de la taula de símbols i el tractament d'errors, actuen amb les sis fases anteriors. En la figura 2.3 observem una descomposició típica d'un compilador.

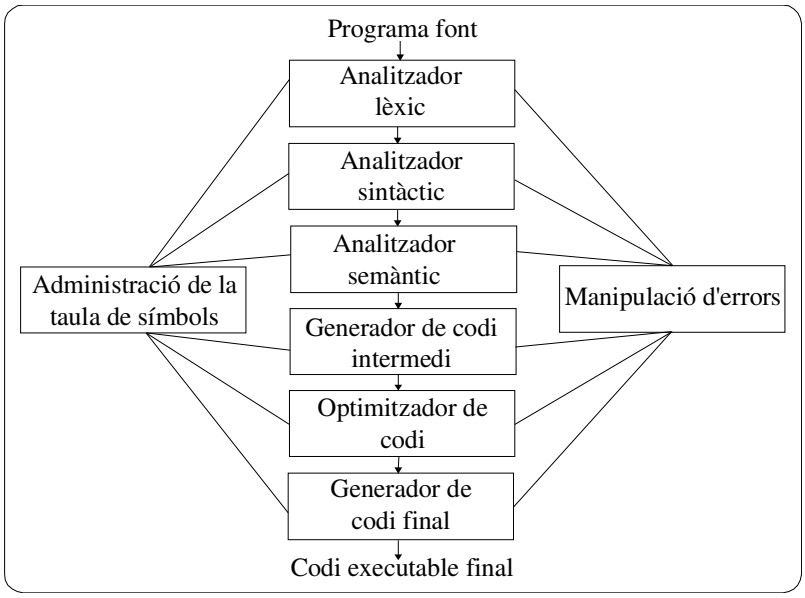


Figura 2.3 Exemple de divisió d'un compilador en *front-end* i *back-end*

2.3 Anàlisi lèxica

L'objectiu principal en aquesta fase és llegir el programa font (una cadena de caràcters emmagatzemada en un fitxer) d'esquerre a dreta, i transformar-lo en una llista de components lèxics anomenats *tokens*. Els espais en blanc i els comentaris s'ignoren. Els *tokens* són una seqüència de caràcters que tenen un significat col·lectiu. Es poden diferenciar en tres tipus importants:

- Paraules claus (corresponen a les instruccions).
- Constants numèriques o literals.
- Símbols (operadors, punt, ...)

Cada *token* té associat un valor lèxic: la seqüència de caràcters que l'identifica. Per exemple, en un *token identificador* el valor lèxic seria pròpiament l'identificador llegit (“Susanna”, “var1”,...),

o en un número, seria la cadena de caràcters que determinen el nombre com ara “23”, “53.45”,... A més del valor lèxic, ha d'emmagatzemar el tipus (paraula clau, constant numèrica, identificador,...) per poder ésser identificat posteriorment per l'anàlisi sintàctica.

Els errors lèxics més comuns són conseqüència de valors lèxics massa llargs (exemple: un identificador de 700 caràcters), o també literals numèrics incorrectes (exemple: 2.2.3). En el procés de compilació, els errors en l'anàlisi lèxica són molt poc freqüents i solen reportar-se mitjançant el mètode pànic (vegeu 2.7 Tractament d'errors).

Per definir els *tokens* s'utilitzen **expressions regulars**. Una expressió regular és una notació molt important per definir *tokens* basada en els anomenats llenguatges regulars. Una expressió regular ha de complir qualsevol dels següents cassos:

1.- E = { }	Denota que és un *token* buit
2.- E = x	Denota un *token* que conté “x”.
3.- E = A | B	Denota que el *token* es pot definir com el *token* A o el B.
4.- E = A · B	Denota que el *token* està format per el *token* A seguit per B. Per abreviar es sol escriure com: A B.
5.- E = A^*	Denota que el *token* està format per cap, una o més vegades el *token* A.
6.- E = A^+	Denota que el *token* està format per A com a mínim una, o més, vegades.

Posem un exemple, un identificador en **pascal** seria:

identificador = (lletra) (lletra | dígit)*

Com es pot observar, comença sempre per una lletra i pot ésser seguida per lletres o dígits. El *token* lletra es pot definir de la següent forma:

lletra = a | b | c | ... | A | B | ... | Z

I un dígit es representa per l'expressió regular següent:

dígit = 0 | ... | 9

Els autòmats finits deterministes són els encarregats de reconèixer els diferents *tokens*. Un autòmat es compon d'un conjunt d'estats, i d'un conjunt de transicions entre aquests. Les transicions són provocades per l'arribada d'un símbol del nostre alfabet (d'un caràcter en el cas de l'identificador). Sempre té un estat inicial i un o més estats finals. Un autòmat finit és determinista si, per a qualsevol parell d'estats en què existeixi una transició, aquesta és única. L'avantatge dels autòmats finits deterministes és que són fàcils de programar-los i, per tant, construir un analitzador lèxic per el nostre llenguatge no és gaire complicat.

2.4 Anàlisi sintàctica

Un analitzador sintàctic agrupa els *tokens* del programa font construint un arbre de derivació o sintàctic que expressa l'estructura del programa font. El compilador utilitza aquest arbre per sintetitzar la sortida.

2.4.1 Gramàtiques lliures de context

Per definir la sintaxi dels llenguatges de programació s'utilitzen les gramàtiques independents de context o notació BNF (Backus-Naur Form). Les gramàtiques permeten saber, donada una paraula, si aquesta pertany o no a un llenguatge. Per exemple, la gramàtica lliure de context **L** defineix el llenguatge dels palíndroms sobre l'alfabet (a,b):

$$L = \{\{a, b\}, \{S\}, S, \{S \to \lambda, S \to a, S \to b, S \to aSa, S \to bSb\}\}$$

Una gramàtica està formada per un conjunt de símbols terminals (en l'exemple {a,b}), una sèrie de símbols no terminals ({S}), un símbol inicial (S) i unes regles de producció:

$$S \to \lambda$$

$$S \to a$$

$$S \to b$$

$$S \to aSa$$

$$S \to bSb$$

Com es pot observar, la part esquerra de la regla és un símbol no terminal que l'identifica, i la part dreta és un conjunt de símbols terminals i no terminals que la defineix. Una gramàtica lliure de

context compleix que, en totes les seves regles de producció, hi ha un únic no terminal a la part esquerre de la producció.

Per comprovar sintàcticament un programa, ha de ser possible construir un arbre de derivació que permeti arribar al símbol inicial, partint de la llista de *tokens* obtinguts a la fase anterior i aplicant-hi les regles de producció.

Existeixen dues estratègies per construir l'arbre de derivació:

- Ascendent: partint dels *tokens* de l'entrada s'ha d'arribar al símbol inicial.

  Ex: ababa abSba aSa S
- Descendent: partint del símbol inicial, s'ha d'arribar als *tokens* de l'entrada.

  Ex: S aSa abSba ababa

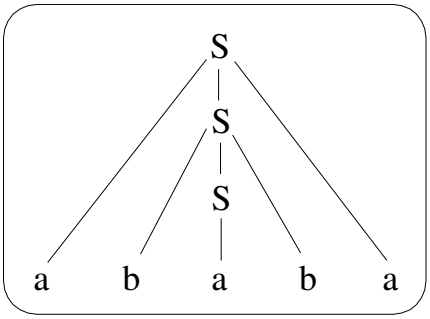


Figura 2.4 Arbre derivació descendent del palíndrom *ababa*

Les gramàtiques lliures de context presenten dos problemes que calen tenir en compte:

1. Ambigüitat. Una gramàtica lliure de context és ambigua si existeix, com a mínim, una paraula del llenguatge definit per la gramàtica que té dos o més arbres de derivació diferents.
2. Indecisió. Una gramàtica lliure de context és indecidible si en arribar al *token* no podem decidir quina regla hem d'aplicar-hi. Aquest cas es sol trobar si en un moment determinat es pot optar per més d'una regla, i per tant hi ha indeterminisme.

2.4.2 Classes d'analitzadors sintàctics

Els analitzadors sintàctics, també coneguts com a *parsers* (anglicisme), es divideixen segons l'estratègia utilitzada per construir l'arbre de derivació.

Se'n distingeixen dues classes:

1. Parser descendent. Són parsers que, partint del símbol inicial i aplicant una sèrie de derivacions, arriben al conjunt de tokens de l'entrada. Les gramàtiques que admeten els parsers descendents són les anomenades gramàtiques lliures de context del tipus LL (lectura de l'entrada d'esquerra a dreta, i derivacions fetes per l'esquerra). Els parsers LL han de tenir unes característiques determinades que fan que no hi hagi ambigüitats ni problemes d'indecisió. Si tenim una gramàtica lliure de context LL(1) és molt fàcil construir un parser descendent recursiu o tabulat que reconegui les paraules de la gramàtica. El **PCCTS** i el **ANTLR** generen parsers LL(k) on k és el número de caràcters de l'entrada que es té en compte a l'hora d'escollir una regla per derivar.
2. *Parser* ascendent. Aquests *parsers* construeixen un arbre acabat per el símbol inicial de la gramàtica, partint dels *tokens* de l'entrada. Aquests *parsers* són anomenats *parsers* LR, perquè llegeixen l'entrada d'esquerra a dreta, i executen les regles de dreta a esquerra. Per tal d'implementar un *parser* LR necessitem, a més del programa, una pila i una taula sintàctica que dirigeixi l'anàlisi. Existeixen diverses classes de *parsers* LR:
   a) *SLR*. Presenta problemes de desplaçament-reducció (*shift-reduce*) que fan que el *parser* no pugui decidir el camí a escollir.
   b) *LR canònic*. Resol el problema d'alguns *shift-reduce* perquè té en compte altres *tokens* que ens donen informació per decidir quin camí hem d'escollir. Té l'inconvenient que es dispara el nombre d'estats de la taula sintàctica.
   c) *LALR*. Incorpora els avantatges de *SLR* i de *LR canònic*, és a dir, resol el problema del *shift-reduce* i té un nombre raonable d'estats. Parteix de LR(1) canònic i intenta agrupar els estats possibles. La eina **bison**, que pertany al projecte GNU, genera parsers LALR i és molt utilitzada junt amb altres utilitats GNU.

2.5 Anàlisi semàntica

Aquesta fase analitza el programa font per intentar trobar-hi errors semàntics. Per treballar utilitza l'estructura jeràrquica, arbre sintàctic o arbre de derivació, determinada per la fase d'anàlisi sintàctica.

La comprovació semàntica s'anomena també comprovació estàtica, per distingir-la de la comprovació dinàmica que es realitza en temps d'execució del programa objecte, com fan per

exemple molts llenguatges *script*, com el **JavaScript**.

Les comprovacions principals que realitza són:

1. *Comprovació de tipus*. Un compilador ha de generar un error si els operands d'una operació són de tipus incompatibles.
2. *Conversió de tipus*. Abans de decidir si dos operands són incompatibles, cal mirar si en convertir-los el tipus, de manera correcte segons el compilador, es possible operar. Un exemple: s'intenta sumar una expressió entera amb una altra real. Per poder operar, cal convertir a tipus real la primera expressió i aleshores es possible sumar-les, ja que les dues expressions tindran tipus real.
3. *Comprovacions d'unicitat*. Depenent del llenguatge font a tractar, es possible que un objecte s'hagi de definir exactament una vegada.
4. *Comprovacions relacionades amb els noms*. Hi ha vegades que un mateix nom ha d'aparèixer dues o més vegades. Per exemple, en C/C++ la definició d'una funció en el fitxer capçalera ha de coincidir amb la del fitxer d'implementació.

2.6 Administració de la taula de símbols

La taula de símbols és una estructura de dades que emmagatzema un registre per cada *token identificador* del programa font. En els registres s'hi emmagatzema informació sobre els diferents atributs dels identificadors. Aquests atributs ens proporcionen informació sobre diferents aspectes lligats a ell, com ara: la memòria assignada a un identificador, el seu tipus, el seu àmbit i, en el cas de noms de funcions, coses com el nombre i el pas de paràmetres (per exemple, per referència) i el tipus que retorna.

2.7 Tractament d'errors

Qualsevol fase del compilador pot trobar errors, i aquests han d'ésser reportats a l'usuari. Després de detectar-ne un, ha de tractar-lo d'alguna manera per poder continuar la compilació, ja que se'n poden detectar més en el programa font. Un compilador que s'atura en trobar el primer error no és gaire útil.

Existeixen tres maneres de tractar els errors:

- **Pànic**. És el mètode més senzill, al trobar al primer error, abandona la compilació. Aquest mètode el solem trobar durant l'anàlisi lèxica.
- **Recuperació**. Al trobar un error intenta restablir el procés de compilació, tot llegint *tokens* de l'entrada i continuar tractant així el programa font. Aquest sol ésser utilitzat durant la fase d'anàlisi sintàctica.
- **Intel·ligent**. Al trobar un error realitza canvis semàntics, de manera intel·ligent per el compilador, de manera que pugui es continuar tractant correctament el fitxer.

2.8 Codi intermedi, optimització i generació de codi final

Com ja s'ha explicat abans, el *front-end* després de les fases d'anàlisi genera el codi intermedi. Aquest codi és el punt d'entrada per l'etapa de *back-end*, la qual es depenent de la plataforma hardware per la qual està dissenyat el compilador. L'especificació de codi intermedi ha d'ésser invariable, d'aquesta manera es poden dissenyar molts *back-end* per diferents plataformes, amb el mateix *front-end*.

El codi intermedi, tot i no ser codi màquina, ha de ser molt proper a aquest, ja que així es podrà realitzar bones optimitzacions i la generació de codi final, serà simple. Per tant, ha de complir que sigui fàcil de produir i, a més, fàcil de traduir a llenguatge màquina. Alguns codis intermedis utilitzats per compiladors són: *The Clarity Mcode (*dissenyat per *Sun Microsystems*), *RTL* i *VAM* (*VSL Abstract Machine*).

Una vegada generat el codi intermedi, s'optimitza, seguint unes regles determinades, com l'eliminació de còpies a registres intermedis innecessàries, localització de variables, etc. En aquesta fase d'optimització és on podem examinar les qualitats de diferents compiladors per el mateix llenguatge. Un compilador que pugui optimitzar molt un codi, de tal manera que produeixi codi que s'executi més ràpid, serà més eficient i per tant molt millor de cara al programador davant de compiladors que produeixin codi més lent per el mateix llenguatge font. Existeix tota una teoria i llibres referent a la optimització, ja que és un procés bastant complicat.

Un cop tenim el codi intermedi optimitzat, ja es pot traduir al llenguatge màquina. Aquest procés sol ser bastant directe si és té un codi intermedi ben dissenyat. Hi ha molts compiladors que enlloc de generar codi màquina directament, generen un fitxer en codi assemblador, el qual serà posteriorment assemblat, sorgint finalment el binari que podrà ésser executat.

En el cas del **Java** i del **Visual Basic 4** i versions anteriors, no generen codi màquina, sinó que finalment obtenim *bytecodes* pel primer i *P-Code* per el segon. Ambdós són llenguatges binaris, bastant propers a codi màquina. Podem pensar que és codi per a una màquina abstracta. Aquest codi ha de ser processat per l'intèrpret a l'hora d'executar-se. L'intèrpret del **Java** és el *Java runtime* (JRE), mentre que el del **Visual Basic 4** és una llibreria dinàmica (DLL) que s'ha de distribuir amb el propi executable. El fet d'haver d'interpretar el codi, fa que el programes generats siguin una mica més lents que els dels compiladors escriuen codi màquina. No obstant, són més ràpid que llenguatges que interpreten directament el codi font, com ara el **JavaScript** o el **basic** original.

2.9 Compilació dirigida per la sintaxi

Fins ara s'ha exposat la visió de que el compilador executa seqüencialment cada una de les fases. Però això no es troba gairebé mai. Normalment, el procés de compilació és dirigit per l'anàlisi sintàctica: l'analitzador sintàctic va demanant *tokens* a l'analitzador lèxic, i el propi *parser* conté codi per l'anàlisi semàntic i generació de codi.

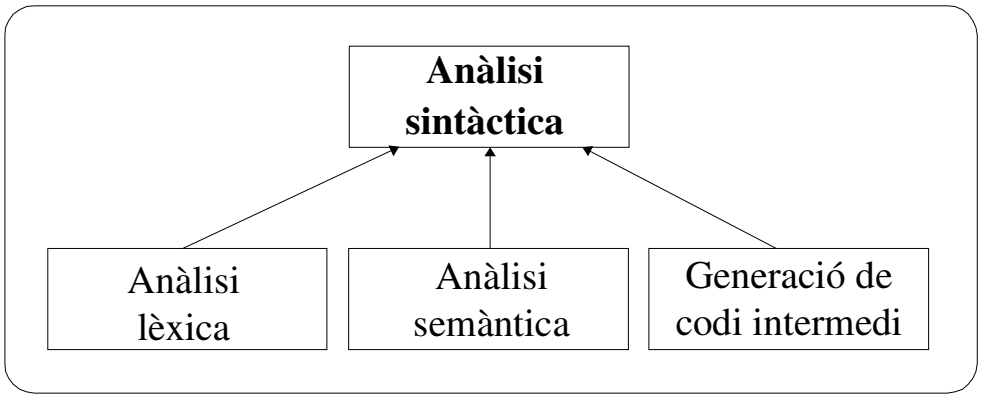


Figura 2.5 Funcionament de la compilació dirigida per la sintaxi

2.10 Marc de treball del traductor implementat

El traductor implementat en el projecte realitza les etapes més importants del procés de compilació, que són: l'anàlisi lèxica, l'anàlisi sintàctica i l'anàlisi semàntica. Les etapes de generació de codi intermedi, optimització i generació de codi final són substituïdes per una fase de generació de codi **Java**.

Les fases implementades realitzen les mateixes comprovacions que faria qualsevol compilador estàndard real. Això vol dir, que si un programa escrit en pseudocodi es tradueix sense que es reporti cap error, segur que el fitxer **Java** de sortida pot ésser compilat directament. Paral·lelament, si al traduir un algorisme es generen errors, també es pot estar segur que la seva corresponent traducció tampoc compilaria.

En la següent figura es pot observar la descomposició en etapes del traductor implementat:

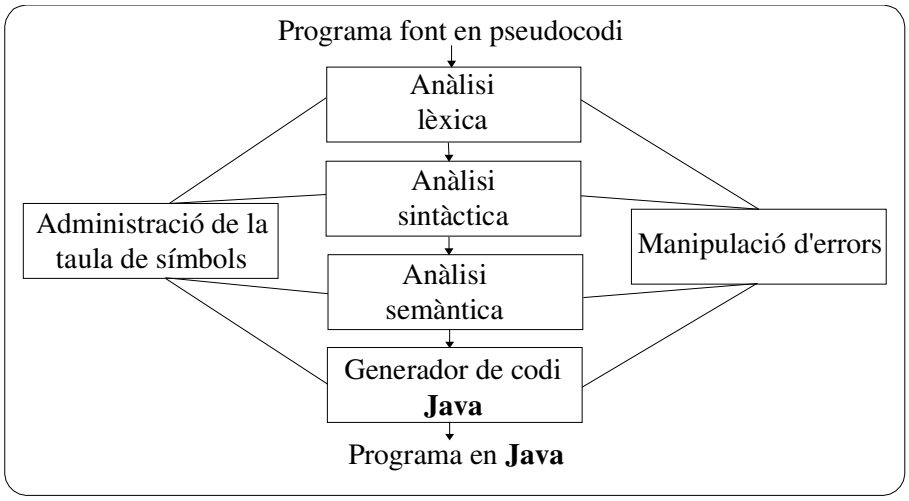


Figura 2.6 Fases del traductor implementat

Cal recordar que la figura mostra l'esquema conceptual en fases ja que, com s'ha explicat anteriorment, el procés de traducció està dirigit per l'anàlisi sintàctica. Aquesta etapa és l'eix central d'execució. S'encarrega de demanar *tokens* a l'anàlisi lèxica, de construir l'arbre de derivació corresponent al fitxer d'entrada, aplicar-hi les comprovacions semàntiques i traduir les sentències al llenguatge **Java**.

3. Pseudocodi 03

3.1 Introducció al pseudocodi

El traductor accepta el llenguatge pseudocodi 03. Aquest està basat en el pseudocodi habitual amb algunes modificacions i afegits, per tal d'explotar el model d'orientació a objectes. Existeixen les instruccions repetitives, condicionals i l'assignació. A més s'afegeix el concepte de classe, polimorfisme, herència i encapsulació, que són els pilars del paradigma de programació amb objectes. La programació s'esdevé més centrada en el disseny de classes i estructures de dades, i no pas en els algorismes, que és on estava centrat el pseudocodi anterior.

Aquest capítol és una referència al llenguatge per tal de tenir una ajuda a l'hora de programar en pseudocodi i treure tot el partit a l'eina desenvolupada. Cal conèixer bé el llenguatge que tracta el traductor per entendre com està definit aquest i com actua. Per tant, es definirà la gramàtica del pseudocodi d'una manera simple, directa i entenedora.

Es farà una visió bastant ràpida i no s'entrarà en gaires de detalls, ja que se suposa uns mínims de coneixement de programació, i no s'explicaran conceptes bàsics com per exemple definir que és una variable. En canvi, s'explicarà com es declara i s'utilitza en el pseudocodi.

Finalment, cal fer notar que el pseudocodi no és sensible a majúscules i minúscules (no és *case sensitive*). Això significa que una variable tota en majúscules i una altra variable tota en minúscules són interpretades com la mateixa variable. Això fa més fàcil la programació per els novells, que solen patir molt per la característica de distincions de majúscules i minúscules del **C++** i del **Java**.

3.2 Conceptes bàsics

A continuació s'enumeraren una sèrie de conceptes bàsics en la definició d'un llenguatge de programació.

- **Comentaris**: línies escrites per el programador que no són processades per el compilador i tenen efecte de documentació.

- **Sentències**: és una línia de programa, normalment anomenada també instrucció.
- **Blocs de codi**: és un grup de sentències que formen una unitat.
- **Paraules reservades**: Paraules que s'han predefinit en el llenguatge pseudocodi 03 i no poden ser utilitzades com a identificadors.
- **Identificadors**: són els noms que es donen a les classes, variables, funcions,... Els identificadors poden ser qualsevol cadena alfanumèrica formada per lletres i números, amb la restricció que el primer caràcter ha d'ésser una lletra o el caràcter guió-baix: "_".
- **Literals**: valors constants que s'escriuen diferent depenen del tipus de dades que representin. Per exemple, és diferent **123** de "123", ja que el primer simbolitza un número enter, mentre que el segon correspon a una cadena.
- **Expressions**: combinació de termes que s'avaluen a un únic valor, com per exemple una suma, o una comparació lògica.
- **Operadors**: serveixen d'unió entre factors i expressions. Poden ser matemàtics com la suma, resta,... o també lògics, com igual, major que,...

Passem a explicar-los més detalladament, i com són tractats en el pseudocodi 03.

3.2.1 Comentaris

Els comentaris són tractats d'igual forma que el **C++**. El primer tipus de comentari és el multilínia, comença amb "/*" i acaba amb "*/".

Per exemple:

```
/* Això és un comentari
de més d'una línia de text */
```

El segon tipus de comentari és aquell que comença amb la marca de comentari "//" i finalitza al final de la línia del fitxer (retorn de carro).

Per exemple:

```
// Aquest comentari arriba només fins a final de línia
```

3.2.2 Sentències

Una sentència és una línia de codi. S'utilitza el retorn de carro com element de puntuació per indicar el final de la instrucció. Es pot utilitzar el punt i coma per escriure més d'una sentència en la mateixa línia.

Per exemple:

```
a <- 2+3; a <- 56-2
```

És el mateix que:

```
a <- 2+3
a <- 56-2
```

Els espais entre les parts d'una sentència poden consistir en qualsevol número d'espais en blanc o tabulacions.

3.2.3 Blocs de codi

Les sentències es solen agrupar dintre de blocs de codi. Els blocs de codi solen estar determinats per un nom que l'identifica al principi, i al final amb una “f” seguida del mateix nom.

Per exemple, el bloc de declaració de variables seria:

```
var
      // Aquí anirien les declaracions de variables
fvar
```

3.2.4 Paraules reservades

Una paraula reservada és una paraula que posseeix un significat especial per el traductor, tal com el nom d'un tipus de dades, una instrucció, entre d'altres.

En la pàgina següent hi ha la taula 3.1 en la que es mostra la llista de paraules reservades.

Paraules reservades			
algorisme	escriure	fmetode	metode_classe
altrament	escriureln	fmetode_classe	metodes
atributs	falgorisme	fmetodes	metodes_classe
atributs_classe	fals	fmetodes_classe	mod
boolea	fatributs	fper	no
caracter	fatributs_classe	fsi	null
cas	fcas	funcio	o
cert	fclasse	fvar	pas
classe	fconst	hereta	per
const	fconstructor	i	real
constructor	fdestructor	implementacio	res
crear	fer	interficie	retorna
de	ffuncio	jo	si
destructor	fimplementacio	llavors	super
destruir	fins	llegir	taula
div	finterficie	mentre	var
enter	fmentre	metode	

Taula 3.1 Paraules reservades del pseudocodi 03

3.2.5 Identificadors

Un identificador és el nom que se li dóna a una variable, classe o funció. Es pot escollir l'identificador que es desitgi, mentre que comenci per una lletra o un guió baix "_" seguit d'una combinació de lletres i números, i han de ser diferents a qualsevol paraula reservada. Existeix un conveni en la nomenclatura dels identificadors, que es segueix en la memòria i en el codi font del projecte, tot i que no és obligatori (vegeu 1.4 Convencions).

3.2.6 Literals i tipus de dades

Si els identificadors són el símbol al que fem referència a un valor, un literal és un valor concret com 39 ó "numeret". Cada literal correspon a un tipus de dades, excepte el literal cadena, que només existeix com a literal, però no es pot declarar cap variable de tipus cadena.

Tipus de dades	Literal	Exemples
enter	Dígits.	10 239
real	Dígits amb un punt decimal.	3012.0 9.382
boolea	**cert** o **fals**	cert fals
caracter	Qualsevol caràcter ASCII entre cometes simples.	'a' 'B'
cadena	Una seqüència de caràcter o seqüència d'escapament entre cometes.	“Hola món\n” “Més proves”

Taula 3.2 Format per els literals de cada tipus de dades

Com ja hem dit, el tipus cadena en sí no existeix, només existeix el literal. Ha sigut introduir per simplificar a l'usuari l'escriptura de missatges per pantalla, ja que és evident que escriure caràcter a caràcter una frase per poder ésser impresa és massa laboriós. No obstant, el tipus cadena o *string* es pot implementar fàcilment en una classe.

3.2.7 Expressions i operadors

Les expressions són combinacions de variables, literals i crides a funcions que són avaluades amb un determinat tipus de dades, que pot ser un número, una cadena o qualsevol altre classe o tipus de dades. Normalment les expressions les trobem en el costat dret en una assignació. Les expressions més simples són variables aïllades o literals, i es poden complicar tant com vulguem.

Exemples d'expressions:

```
33
'a'
734*(97-(82+1)/2)
```

Els operadors són símbols que combinen les expressions simples per formar expressions més complexes o apliquen una transformació a una variable. Els operadors poden ser binaris, si són aplicats a dues expressions, i unaris si només són aplicats a una. Per exemple, la multiplicació és un

operador binari, ja que sempre es multipliquen dos números, mentre que l'operador lògic **no** és un operador unari, aplicat només a una expressió de tipus lògica. Quan s'utilitza varis operadors en una expressió, és important saber en quin ordre s'avaluaran. Com en matemàtiques, existeix una *prioritat d'operadors*, i si els operadors tenen la mateixa prioritat, s'avaluen d'esquerre a dreta. Cal recordar que es pot fer ús dels parèntesis per agrupar expressions i definir d'aquesta manera en quin ordre s'han d'avaluar. D'operadors hi ha de tres tipus: operadors aritmètics, operadors lògics i l'operador assignació.

Els operadors matemàtics són els utilitzats en operacions elementals, a la següent taula es pot comprovar els operadors i el seu ordre de preferència.

Operador	Descripció	Preferència
+,-	Més unari,menys unari	1 (major preferència)
*,/	Multiplicació,divisió real	2
mod	Resta (mòdul de la divisió)	2
div	Divisió entera	2
+,-	Suma,resta	3

Taula 3.3 Operadors aritmètics

Els operadors lògics comparen dues expressions per determinar si són iguals, si una és més gran que l'altre,... És aconsellable l'ús de parèntesis per tal de no equivocar-se a fer ús de l'expressió.

Operador	Descripció	Preferència
<,<=,>,>=	Verifica magnituds relatives	4
=,<>	Verifica la igualtat i la desigualtat.	5
i	I condicional	6
o	O condicional	6

Taula 3.4 Operadors lògics/relacionals

3.3 Tipus de dades: declaració de variables i instanciació

En pseudocodi 03 existeixen dos grans tipus de dades: elementals, o també anomenats bàsics, i els de referència. Els elementals són aquells integrats en el llenguatge, i que ja hem vist

anteriorment: **enter**, **real**, **caràcter**, **booleà**. Els tipus de referència són les variables declarades com a qualsevol tipus definit per l'usuari en una classe. Són variables que referencien la instància de la classe (que ha de ser creada posteriorment). Dintre del tipus referència també es troben les taules, que ja s'explicaran més endavant.

Per la declarar una variables s'utilitza la notació del nom de la variable, seguida per dos punts i el tipus. El nom de la variable ha d'ésser un identificador vàlid. En una mateixa línia es poden declarar més d'una variable del mateix tipus, tot separant-les per comes.

Per exemple:

```
variable1: enter                    // Tipus elemental (enter)
var1:enter                          // Tipus elemental (enter)
variable2,variable3: Classe1        // Referència a una classe (Classe1)
```

Les variables objecte, les que són referència a una classe, tenen un tracte una mica especial. A diferència dels tipus bàsics, no poden ser utilitzades directament un cop declarades, sinó que cal *instanciar-les*. Instanciar una classe significa crear-la: es reserva l'espai de memòria que necessita, i pot ser utilitzada per el programa. No es pot utilitzar una classe sense haver-la instanciat prèviament, ja que la referència és nul·la i es produirà un error en temps d'execució. La instanciació d'una variable es realitza amb la paraula reservada **crear** seguida del tipus que es vol instanciar i de parèntesis i opcionalment paràmetres, com si fos una crida a una funció.

Per exemple:

```
variable2 <- crear Classe1()
```

Les constants són dades que no varien en el temps, són dades constants (valgui la redundància). Per tant, una vegada que s'ha declarat la constant i el seu valor, aquest persisteix immutable fins al final del programa. Per definit una constant, es fa servir el següent esquema:

NOM_CONSTANT : tipus_elemental = valor

Les constant només es poden declarar de tipus bàsic (o elemental), i, lògicament, el valor que prenen ha de concordar amb el tipus de constant declarada.

Per exemple:

```
PI: real = 3.1416     // Correcte.
JULIOL: enter = 6.0  // Això provocaria un error, ja que la constant és de
                     tipus entera, i el valor és de tipus real.
```

3.4 Instruccions acceptades

A continuació s'exposaran les instruccions que formen part de la definició del pseudocodi 03, fent notar detingudament quin és el seu format d'ús. Com anteriorment s'havia comentat, el salt de línia és el separador entre les instruccions, per tant, s'ometrà en aquelles instruccions que el necessitin en el seu format, ja que es podrà apreciar visualment en la seva definició.

3.4.1 Instrucció d'assignació

L'operador d'assignació és la fletxa dirigida a l'esquerra ("<-"). La instrucció d'assignació es composa de la part dreta, que és el valor que es vol assignar, i de la part esquerra, que és el destí final del que s'assigna. A la part esquerra hi ha d'haver una variable, que permeti poder emmagatzemar un valor, mentre que la part dreta pot ser qualsevol expressió, això si, del mateix tipus que la variable a la qual es vol assignar.

Per exemple:

```
variable1 <- 3 + 527
variable2 <- variable3
```

L'assignació múltiple present en el pseudocodi 03 no s'ha pogut implementar, a causa de les limitacions de temps i del propi llenguatge **Java**.

3.4.2 Instrucció *res*

No fa res, serveix per simbolitzar que no cal executar cap instrucció arribat a aquell punt.

3.4.3 Instruccions alternatives: *si* i *cas*

Les instruccions alternatives són molt importants ja que permeten desviar el flux d'execució del programa, podent obtenir algorismes molt més interessants, ja que poden tenir diferent resposta a diferents entrades.

El **si** permet avaluar una condició, i executar un bloc d'instruccions si aquesta es compleix, i opcionalment executar un altre bloc d'instruccions diferents si no es compleix. Es pot escriure en dos variants: una línia o multi-línia.

- Una línia:
  si condició **llavors** instrucció **[altrament** instrucció**] fsi**
- Multi-línia:
  si condició **llavors**
  instruccions
  [**altrament**
  instruccions]
  fsi

En el **si** només es pot executar un dels dos codis, ja que o bé es compleix la condició, o bé no es compleix (condició booleana). Com es pot comprovar, la línia de l'**altrament** és opcional, i si no hi és i no és compleix la condició, s'executarà **res**.

Posem un exemple per clarificar-ho:

```
si (a1 < 2) llavors
      b1 <- cert     // Això només passa si a1 < 2
altrament
      si (a2=3) llavors b1<-cert fsi
      // El programa sempre arriba aquí si a1>=2
      // Posarà b1 a cert si es compleix també que a2=3
fsi
```

La instrucció **cas** és semblant a la **si**, ja que ambdues són alternatives, però amb la diferència que la primera permet avaluar múltiples condicions alhora. El format és el següent:

cas
[] condició_1 -> instruccions

[] condició_2 -> instruccions
...
[] condició_n -> instruccions
fcas

Al començament de cada condició hi ha un obrir (**[**) i tancar (**]**) corxet, i són obligatòries, ja que es necessita un delimitador per la condició (sinó la gramàtica que defineix el llenguatge esdevindria ambigua). En el **cas** s'ha d'especificar com a mínim una condició.

Per exemple:

```
// Traducció a cas de l'exemple posat per la instrucció si
cas
[] (a1 < 2) -> b1<-cert
[] (a1 >= 2) -> cas
                [] (a2=3) -> b1<-cert
                fcas
fcas
```

3.4.4 Instruccions iteratives: *mentre* i *per*

Les instruccions iteratives permeten escriure blocs de codi que seran repetits una sèrie de vegades. Això dóna molta potència a l'hora de crear algorismes molt complexos.

La instrucció **mentre** conté una condició, la qual s'avalua la primera vegada abans d'executar les instruccions del bloc, i a cada pas de la iteració es torna a avaluar. Això es va repetint fins que la condició esdevé falsa. Acte seguit el programa salta a la instrucció següent després del **mentre**. La instrucció **mentre** també es presenta en dos variants: una línia o multi-línia.

- Una línia:
  mentre condició **fer** instrucció **fmentre**
- Multi-línia:
  mentre condició **fer**
  instruccions
  fmentre

Per exemple:

```
// Iteració cap endavant
var<-1
mentre (var1 < 10) fer
```

```
        variable1 <- var1
        var1 <- var1 + 1
fmentre
```

En canvi, la instrucció **fer** presenta una iteració no condicional, en la qual el nombre de iteracions que es realitzaran sobre el codi ve definit en la pròpia instrucció. La instrucció **fer** també es presenta en dos variants: una línia o multi-línia, vegem-ne el format:

- Una línia:
  per variable **<-** expressió **fins** expressió **[pas** expressió**] fer** instrucció **fper**
- Multi-línia:
  per variable **<-** expressió **fins** expressió **[pas** expressió**] fer**
  instruccions
  fper

La variable és la que controla el bucle. S'inicialitza amb el valor de la primera expressió, que ha de ser entera, i el bucle itera mentre el valor de la variable sigui més petit que el de la segona expressió, que també ha de ser entera. Com es pot observar, el **pas** és opcional, que significa els increments que es sumen a la variable en cada iteració. L'expressió del **pas** pot ser positiva o negativa, però sempre un nombre enter. Si és negativa, a cada volta de la iteració disminueix el valor de la variable controladora. Si no s'especifica cap **pas**, es pren com a defecte *+1*, i per tant, a cada iteració, la variable de control s'incrementaria en una unitat.

Vegem un exemple:

```
// Iteració cap enrere
per var1<-10 fins 1 pas -1 fer
      variable1 <- var1
fper
// Iteració cap endavant
per var1<-1 fins 10 fer
      variable1 <- var1
fper
```

3.4.5 Instrucció *crear*

La instrucció **crear**, retorna una instància d'un objecte (classe), de tal manera que assigna la memòria que necessita l'objecte.

El seu ús és el següent:

nomVariable <- **crear** tipus(paràmetres)

paràmetres: són paràmetres que pot necessitar-se per instanciar-se el tipus, pot no necessitar-ne cap. La instrucció **crear** crida al constructor de la classe que es vol instanciar, i per tant, els paràmetres han de coincidir amb la definició d'algun dels constructors de la classe (per a més informació referent als constructors consultar 3.6.5 Constructors i destructors). Qualsevol variable de tipus objecte cal que sigui instanciada abans d'ésser utilitzada. Una variable no instanciada té una referència nul·la, que es representa amb la paraula reservada **null**.

Exemple:

```
// crear sense cap paràmetre
variable2 <- crear Classe1()
// crear amb paràmetres
variable2 <- crear Classe1(23,'a')
```

3.4.6 Instrucció *destruir*

Aquesta instrucció destrueix l'objecte, és a dir, allibera la memòria reservada per l'objecte en la instrucció **crear**. Explicat més detingudament, aquesta instrucció crida al destructor de la classe, que s'encarregarà d'alliberar la memòria reservada per el constructor. Es pot dir que és justament la inversa de la instrucció **crear**.

3.4.7 Instrucció *retorna*

Aquesta instrucció fa que es finalitzi l'execució d'una funció o mètode i es retorni el valor.

3.4.8 Instruccions d'E/S

Ja que el pseudocodi ha d'ésser el més senzill possible, s'ha optat de proporcionar al propi llenguatge d'instruccions bàsiques d'entrada i sortides de dades: **escriure**, **escriureln** i **llegir**.

- **escriure**(*expressió*)

Escriu per la sortida estàndard (normalment la pantalla), l'expressió o variable. Si la variable és de tipus elemental, escriu directament el seu valor, si és un objecte, escriu el que retorna directament la funció: **toString**(). Nota: tota classe creada per l'usuari té una funció **toString()** per defecte que pot ser sobreescrita per l'usuari.

- **escriureln**(*expressió*)

Es comporta d'igual manera que la funció **escriure**, però afegeix automàticament un salt de línia al final del que s'escriu, d'aquesta manera, després d'haver escrit l'expressió, es salta a la següent línia.

A **escriure** i **escriureln** se li poden passar literals de tipus cadena, de tal forma que es pot escriure una frase per pantalla. A més, es poden passar diversos caràcters d'escapament. Les seqüències d'escapament són les mateixes que les que suporta el **Java**, a la taula 3.5 es presenten les més importants.

Literal	Descripció
'\b'	Retrocés
'\t'	Tabulador
'\n'	Avanç de línia
'\f'	Avanç de pàgina
'\r'	Retorn de carro
'\\'	Barra invertida

Taula 3.5 Seqüències d'escapament especials

- **llegir**()

Espera que l'usuari entri una dada i la retorna, de tal manera que pot ésser assignat a una variable. Si l'usuari entra un valor d'un tipus que no correspon amb el que pertoca a l'assignació, es llançarà una *excepció* i el programa finalitzarà. La instrucció només retorna dades de tipus elemental, si es fa una assignació a un altre tipus generarà un error en temps de compilació.

3.5 Esquema d'algorisme

Els **algorismes** simbolitzen el punt d'entrada al programa, la funció principal. És com la funció *main* del **C/C++** o el mètode estàtic *main* del **Java**. Degut això, dintre d'un algorisme no es permet la declaració de funcions i/o mètodes de cap tipus. Això és degut a que en les tècniques de programació orientada a objectes, l'algoritme principal serveix només per instanciar els objectes bàsics i eventualment d'entrada/sortida d'algun valor de l'usuari. Per tant, tota la feina corre a càrrec de les classes.

L'estructura d'algorisme és la següent:

algorisme nomAlgorisme
const
NOM_CONSTANT : tipus_elemental = valor
fconst
var
nomVariable: tipus
fvar

instruccions
falgorisme

Les parts de declaració de constants i de variables són opcionals. S'ha mostrat també com s'introdueixen la declaració de constants i variables, explicades en el capítol anterior, dintre d'un algorisme, i amb el que s'ha vist fins ara, es poden fer algorismes senzills.

Com ja s'ha vist, dintre els operadors aritmètics no n'hi ha cap que simbolitzi la potència. Mostrarem un programa exemple, que demanarà la base i l'exponent a l'usuari, i calcularà la potència, és a dir, elevarà la base a l'exponent especificat.

Programa potència:

```
algorisme potencia
var
      base,exp,result,num: enter
fvar
```

```
        escriure("Escriu la base: ")
        base <- llegir()
        escriure("Escriu l'exponent: ")
        exp <- llegir()

        si (exp=0) llavors
                  // Qualsevol nombre elevat a zero és 1
                  result <- 1
        altrament
                  // Inicialitzarem el resultat amb la base
                  result <- base
                  // Farem tantes multiplicacions com indiqui l'exponent
                  per num<-2 fins exp fer
                            result <- result * base
                  fper
        fsi

        escriure("El resultat: ")
        escriureln(result)

    falgorisme
```

Com s'ha pogut observar, en el programa s'ha utilitzat instruccions de molts tipus: iteratives, condicionals, d'E/S,... variables i constants. I a més, és plenament funcional. Amb aquest esquema es poden resoldre problemes petits, però no n'hi ha prou. Per resoldre problemes una mica "seriosos" ens cal alguna cosa més, que ens ajudi a estructurar i desenvolupar programes molt llargs. Necessitem fer ús de les classes.

3.6 Classes

El pseudocodi nou ha d'afavorir la bona programació orientada a objectes i ha de proporcionar un llenguatge senzill per poder crear aquests objectes. Per designar-los farem servir les classes, que és el concepte més estès gràcies al **C++** i al **Java**. Una classe és la definició d'un objecte, que conté unes dades que el defineixen, anomenades atributs, i unes accions que permeten que actuï i es comuniqui amb els altres objectes, anomenades mètodes. Com que la classe és només una definició, quan es declara una variable de tipus objecte, cal instanciar-la per poder ésser utilitzada, com ja s'ha vist anteriorment.

Per definir una classe s'utilitza la paraula reservada **classe** seguida d'un identificador, i al final del bloc: **fclasse**.

Per tant, una classe restaria d'aquesta forma:

```
classe PprimeraClasse
      ...
fclasse
```

3.6.1 Encapsulament

L'encapsulament és una propietat intrínseca de l'orientació a objectes que consisteix en establir unes variables, funcions i atributs privats, que només poden ésser accedits des de dintre la classe, i uns mètodes i atributs públics que poden ser accedits des de fora de la classe. Aquests últims permeten alterar-ne les propietats i comportament. Per a poder separar les dades públiques de les privades, hi ha el bloc **interficie** i el bloc **implementacio**.

A la interfície es declara tot allò que és públic, i per tant pot ser accedit des de fora la classe, és el que defineix com es relaciona amb les altres classes. En aquesta secció només hi ha declaracions, però no hi ha codi de cap tipus. En canvi en el bloc d'implementació hi ha el codi, o implementació, dels mètodes definits en la secció interfície, i a més també hi ha altres funcions privades, així com variables internes a la classe, declarades dintre un bloc **var-fvar**.

Així, un primer esquema d'una classe seria:

```
classe PrimeraClasse
      interficie
                // Aquí va tot allò que sigui públic i es vulgui
                // que sigui visible a altres classes externes.
      finterficie
      implementacio
                // Aquí va tot el codi de la classe, així com dades que
                // no volem mostrar o que siguin accedides des de fora.
      fimplementacio
fclasse
```

3.6.2 Atributs i mètodes

Els atributs són variables públiques, i per tant, visibles des de fora la classe. Podem definir com a atribut qualsevol tipus de variable. Els atributs sempre estan definits en el bloc **atributs** dintre la **interficie**. Al ser variables, els atributs es declaren dintre un bloc **var**.

Els mètodes són el que s'anomenava accions i funcions en el pseudocodi habitual. Els mètodes poden retornar o no un valor. A la secció **interficie** trobem declarades les capçaleres dels mètodes dintre el bloc **metodes**. La capçalera d'un mètode és:

metode identificador **(***paràmetres_formals)* [**retorna** identificador: tipus]

Paràmetres formals indica els paràmetres que s'han de passar al cridar al mètode. Han d'estar definits com una llista de: identificador **:** tipus. Si n'hi ha més d'un, han d'estar separats per comes.

El **retorna**, és opcional. Si hi és, especifica que el mètode es comporta com una funció, i retorna un valor del tipus definit. L'identificador declarat en el **retorna** és el paràmetre de retorn.

El codi del mètode està dintre el bloc **implementacio**: només cal tornar a escriure la capçalera del mètode, escriure el codi, i acabar amb un **fmetode**. L'identificador de retorn declarat en el **retorna** es declara automàticament en la implementació del mètode. Es pot utilitzar directament, no cal declarar-ho explícitament.

Els atributs i mètodes sempre estan lligats a una instància, i per tant, cada instància tindrà valors diferents per els atributs i els seus mètodes retornaran valors diferents, depenent de les dades internes que tingui emmagatzemades.

3.6.3 Atributs de classe i mètodes de classe

Els atributs i mètodes pertanyen a una instància, però ens pot interessar que hi hagi dades que pertanyin a la pròpia classe, com per exemple, el número d'instàncies creades, diferents constants (com ara PI, nombre E,... per una classe matemàtica), entre d'altres. Les variables, constants i

mètodes que vulguem que siguin de la classe, es declaren en el bloc **atributs_classe** per les constants i variables, i en el bloc **metodes_classe** per els mètodes. La declaració d'un mètode de classe es fa de la mateixa manera que la d'un mètode d'instància, amb la diferència que s'ha d'utilitzar la paraula reservada **metode_classe**, enlloc de **metode**. La implementació del mètode de classe és també igual a la del mètode, canviant les paraules reservades a **metode_classe** i **fmetode_classe**, per començar i acabar el bloc respectivament.

Com es pot comprovar, no s'ha comentat que es puguin declarar constants en un bloc **atributs** d'una instància. El propi llenguatge no ho admet, ja que una constant que pertanyi a una instància no té sentit, les constants són pròpies d'una classe.

En els mètodes de classe, només poden accedir als atributs de classe, i no pas als atributs d'instància, ja que el mètode de la classe no sabria quin valor d'instància agafar per l'atribut.

Resumint, fins ara, la declaració completa d'una classe podria ser:

```
classe PrimeraClasse
      interficie
                // Aquí va tot allò que sigui públic i es vulgui
                // que sigui visible a altres classes externes.

      atributs_classe
                const
                          // Constants de la classe.
                fconst
                var
                          // Variables que representen els atributs
                          // d'una classe.
                fvar
      fatributs_classe
      atributs
                var
                          // Aquí estan definits els atributs
                          // que pertanyen a cada instància.
                fvar
      fatributs
      metodes_classe
                          // Els mètodes pertanyents a la classe.
                metode_classe sumar_instancia() retorna e: enter
      fmetodes_classe
```

```
        metodes
                  // Els mètodes pertanyents a cada instància
                  metode fer_alguna_cosa(num: enter)
        fmetodes
        finterficie
        implementacio
                  // Aquí va tot el codi de la classe, així com dades que
                  // no volem mostrar o que siguin accedides des de fora.
                  metode_classe sumar_instancia() retorna e: enter
                            // Implementació del mètode de classe
                  fmetode_classe
                  metode fer_alguna_cosa(num: enter)
                            // Implementació del mètode d'intància
                  fmetode
        fimplementacio
  fclasse
```

L'ordre de la declaració de la interfície: atributs de classe, atributs, mètodes de classe i mètodes no es pot alterar, tot i que tots són opcionals.

3.6.4 Atributs i funcions privades

Per poder permetre l'encapsulament en els objectes, hem de proporcionar un mitjà per tal de definir atributs i funcions que no siguin accessibles des d'objectes externs. D'aquesta manera proporcionem coherència a les dades, ja que sabem que cap element extern hi té accés, i per tant no poden ésser modificades per ningú més que la pròpia classe.

Els atributs o variables privades s'han de declarar en el seu corresponent bloc **var** dintre el bloc d'implementació, just abans de la implementació del codi dels mètodes i funcions. Les funcions privades estan declarades amb la paraula **funcio** i només les trobem en el bloc implementació i no en a la interfície (sinó no serien privades).

Per exemple, declararem un atribut privat de tipus enter, i una funció privada que retorna el número multiplicat pel paràmetre passat:

```
  implementacio
        var
                  num: enter
        fvar
```

```
        funcio mult2(factor: enter) retorna e: enter
                  e <- num * factor
                  retorna e
        ffuncio
    fimplementacio
```

3.6.5 Constructors i destructors

En les classes existeixen dos mètodes especials anomenats: **constructor** i **destructor**. El **constructor** es executat automàticament quan es crea una instància de la classe, és a dir, quan s'utilitza la instrucció **crear**. Serveix normalment per inicialitzar algunes variables o reservar memòria. El **destructor** es executat automàticament quan es crida a **destruir**, i normalment s'executa codi per alliberar la memòria reservada en el constructor, o en altres mètodes executats per la classe. Els constructors estan declarats en el bloc d'interfície encara que no pot ser cridat directament des de fora la classe, ja que, com ja s'ha dit, es cridat automàticament per la instrucció **crear**. En canvi, els destructors només apareixen en el bloc d'implementació.

Cal dir que és millor no escriure codi que sigui essencial executar-se al destruir la instància de la classe, com un alliberament dels recursos. Això és així per que el llenguatge destí al que es tradueix, el **Java**, no posseeix estrictament un destructor, sinó que defineix la funció membre **finalize** que actua de manera semblant. El control de memòria del **Java** recau en el *Garbage collector* (en endavant GC), a diferència del **C++** que s'ha d'implementar tot per el programador. Això significa que quan un objecte no s'utilitza, la memòria d'aquest és alliberada per el GC, i per tant, perd importància el destructor com alliberament de la memòria reservada. El mètode **finalize** s'executa just abans que el GC destrueixi l'objecte, però no es pot confiar plenament en l'execució del mètode. No sabem amb exactitud quan el GC tractarà l'objecte, i quins objectes seran tractats abans que els altres; és més, pot ser que el mètode no s'executi mai per què el programa ha finalitzat abans. Per tant, si s'ha d'alliberar recursos (fitxers oberts, sockets,...) al destruir la instància de la classe, és millor programar un mètode a part, per exemple *tancar()*, que s'encarregui de l'alliberament i cridar-lo explícitament des del programa.

Per declarar els constructors i els destructors, es segueix la mateixa pauta que amb els mètodes, però canviant la paraula reservada **metode** per **constructor** i **destructor**, respectivament. Els destructors no poden portar cap paràmetre, mentre que els constructors si accepten paràmetres, que es passen quan es fa la crida a **crear**.

Els constructors apareixen sempre en la interfície, per què són els mètodes que es criden implícitament amb la instrucció **crear**, i poden acceptar diferents paràmetres que s'han d'especificar. Tot i que els destructors es criden també implícitament amb la instrucció **destruir**, al no poder passar cap paràmetre, no es declaren en la part d'interfície.

3.6.6 Sobrecàrrega

També coneguda com polimorfisme o funcions polimòrfiques, significa la possibilitat de declarar un mètode una o més vegades, sempre que s'utilitzin paràmetres diferents per cada una. El conjunt de paràmetres d'un mètode o funció s'anomena signatura, i per tant, la sobrecàrrega permet tenir dos mètodes amb el mateix nom i diferent signatura. Si dos funcions tenen els paràmetres iguals, però diferent paràmetre de retorn, no es pot sobrecarregar, ja que la signatura només la formen els paràmetres formals, i es donarà un error en temps de compilació.

Per exemple, es podria sobrecarregar la funció suma d'una classe, per que es pogués sumar tant enters com reals:

```
metode suma(e: enter)
      // Suma l'enter a la classe
fmetode
metode suma(r:real)
      // Suma el real a la classe
fmetode
```

3.6.7 Herència

La herència és un dels mecanismes més potents del paradigma de programació orientada a objectes. Permet que una classe *hereti* automàticament els mètodes i atributs de la seva classe pare, anomenada també superclasse. Adicionalment, fa que qualsevol funció que tingui un paràmetre de

tipus de la classe pare, accepti també qualsevol classe filla, anomenada també subclasse.

Per definir que una classe hereta d'una altra, o que simplement és una subclasse, s'utilitza la paraula reservada **hereta** seguida d'un identificador que determina la classe pare, just al costat de la definició de la classe:

classe nomClasse **hereta** nomClassePare

...

fclasse

Els mètodes i funcions heretades, es poden sobreescriure en la classe filla, ja que com que les dades que emmagatzema la classe filla poden ser diferents que la del pare, també pot variar la implementació dels mètodes.

L'herència permet definir una estructura de classes jeràrquica, i ajuda a la reutilització de codi, ja que hi ha funcions que seran programades un sol cop en la classe pare, i les classes filles l'heretaren i en podran fer ús sense haver de tornar a implementar-les.

3.6.8 Les referències *jo* i *super*

Aquestes dues paraules reservades són referències a la pròpia classe i a la classe pare respectivament. Dintre una classe, et pots referir a un atribut o mètode posant a davant **jo**, seguit d'un punt:

jo.nomAtribut

La paraula **super** serveix per referir a un mètode o atribut de la classe pare. Això és molt útil, ja que permet accedir a mètodes de la classe pare, quan existeixen mètodes sobreescrits en la subclasse. Per accedir a un mètode de la superclasse, només cal escriure **super**, seguit de punt i l'atribut o mètode:

super.nomMetode(paràmetres_actuals)

La crida de **super** simplement, significa la crida al constructor de la classe pare:

super()

o bé amb paràmetres:

super(paràmetre1,paràmetre2,...)

3.7 Taules

Una taula és un grup de variables del mateix tipus, al qual ens podem referir mitjançant un nom en comú. El tipus pot ser tan un tipus bàsic, com l'enter, com qualsevol classe definida per l'usuari. Una taula és considerada com una classe, però degut al seu habitual ús, està dotada d'una sintaxi especial.

La seva declaració és:

nomTaula: **taula**[] de tipus

nomTaula es comporta com una referència a un objecte, i per tant, abans de poder ésser utilitzat, s'ha de instanciar, de la següent manera:

nomTaula <- **crear** tipus[*número*]

on número és un enter que indica la grandària de la taula.

Per accedir a cada element de la taula, s'indexa d'igual manera que es fa amb **C++** o **Java**:

nomTaula[índex]

on índex ha d'ésser una expressió de tipus entera.

Les taules poden ser també multidimensionals, i es tracta també d'igual manera que el **Java** i el **C++**:

nomTaula: **taula**[][] de tipus // Dos dimensions

nomTaula: **taula**[][][] de tipus // Tres dimensions

I a l'hora d'instanciar la taula, cal indicar la grandària de cada dimensió. Cal notar, que les dimensions poden tenir diferent grandària, per exemple:

```
var
      taula1: taula[][] de enter
fvar

taula1 <- crear enter[5][2]
```

Per poder saber el límit de la taula, podem fer servir el mètode **longitud()** que retorna el límit superior, és a dir, la llargària de la taula. Per cridar-lo, cal posar el nom de la taula, seguit d'un punt i la crida:

nomTaula.**longitud()**

Els dos mètodes retornen un enter. Cal notar, que el límit inferior és sempre zero, ja que s'utilitza el mateix format que el **Java**, mentre que **longitud()** retorna la grandària de la taula amb la qual s'ha instanciat. Si té més d'una dimensió, haurem d'accedir a cada una de les dimensions per saber la seva grandària.

Finalment, cal apuntar un error freqüent entre els programadors novells: si la taula està formada per objectes, enlloc de elements de tipus elemental, una vegada que s'ha instanciat la taula, no es pot accedir directament als elements. Cal primer, crear l'element, ja que com s'ha explicat, els objectes cal instanciar-los abans d'utilitzar-los. Vegem un exemple per clarificar la qüestió:

```
var
      taula1: taula[] de Classe1
fvar

taula1 <- crear Classe1[10]     // Instancia la taula amb 10 elements
taula1[0] <- crear Classe1()    // Instancia el primer element de la taula
taula1[0].metode1()            // Accedeix a un mètode del primer element
taula1[1] <- crear Classe1()    // Instancia el segon element de la taula
      ...
```

3.8 Ús de les classes

Per utilitzar una classe externa, només cal declarar-la com si fos un tipus elemental. El compilador, si no troba la definició en la taula de símbols, intenta carregar el fitxer i llegeix la

interfície d'aquest.

S'ha pogut veure anteriorment que l'operador punt és el que ens permet accedir als atributs i mètodes d'una classe. Per tant, si volem accedir a un mètode només cal posar el nom de la variable, seguit d'un punt i la crida al mètode pertinent. Lògicament només es pot accedir als atributs i mètodes públics, aquells que han estat declarats en el bloc d'interfície.

Recordem que els atributs i mètodes poden ésser d'instància o de la classe. Si són d'instància, els més comuns, fan referència a les dades de la pròpia instància, i són cridats externament:

nomInstanciaDeClasse.metode1(*paràmetres_actuals*)
nomInstanciaDeClasse.atribut1

En canvi si són de classe, no estan lligats a una instància en concret, sinó que són més generals. Podem pensar, per exemple, en la contant PI, o un mètode de comparació de dos enters. En aquest cas, per cridar-los ens referirem al nom de la classe, enlloc del nom de la instància:

NomClasse.metodeClasse1(*paràmetres_actuals*)

NomClasse.atributClasse1

NomClasse.CONSTANT1

3.9 Format del fitxer font

Els fitxers font de pseudocodi 03 tenen per defecte l'extensió ".pse". En un fitxer hi pot haver-hi un algorisme o una classe. Les definicions de classe i algorisme s'han vist en els capítols anteriors. Quan s'assigna un nom a la classe o algorisme, és convenient (però no necessari) que sigui igual al nom del fitxer. Ja que, el fitxer de sortida que genera el compilador té el mateix nom que el fitxer origen, canviant l'extensió per ".java". I quan es passa el compilador del **Java** es generen les classe binàries en *bytecode*, i aquests fitxers si tenen el mateix nom que la classe declarada en el fitxer font. Per tant, poden diferir del nom del fitxer **Java** o del pseudocodi original. Per això, s'aconsella que la classe/algorisme tinguin el mateix nom que el fitxer, ja que així el fitxer **Java** i el fitxer de classe final tindran tots tres el mateix nom (naturalment diferent extensió). D'altra manera, sorgirien fitxers amb noms diferents i la situació podria esdevenir en un petit caos de noms.

3.10 Restriccions del traductor

El traductor posseeix unes restriccions que no el fan cent per cent compatible amb el pseudocodi.

Primerament, la visibilitat dels atributs i mètodes és tractada com el **C++**, ha de ser primer declarats per poder ésser utilitzats/cridats. El **Java** permet referències endavant, és a dir, cridar un mètode que està declarat més endavant, mentre que el traductor no.

Finalment, no s'accepta l'assignació múltiple, ja que no ha estat implementada. Això no és un gran problema, per què les instruccions amb assignacions múltiples es poden desglossar ràpida i fàcilment en instruccions amb assignacions simples.

4. Entorn de treball

4.1 Eines per la generació de compiladors

Una eina de creació de compiladors o traductors normalment es divideix en dues fases: la creació d'un analitzador lèxic, que a partir de la definició de les expressions regulars permet detectar els *tokens*; i la creació d'un analitzador sintàctic, a partir d'una gramàtica d'entrada per part del programador. Els analitzadors lèxics i sintàctics no són difícils de construir, algorísmicament parlant, el problema radica que per una gramàtica de mida mitjana, el programa pot ser de l'ordre d'uns quants milers de línies, i la possibilitat d'introduir errors és bastant elevada. Si es produeixen modificacions a la gramàtica, s'ha de modificar el programari que l'analitza. Això incrementa el temps i el cost de manteniment, així com fa aparèixer el problema que qualsevol error introduït en la modificació de l'algorisme esdevindrà en què fallarà l'etapa d'anàlisi.

Al ser un procés molt fàcil d'automatitzar, normalment s'utilitzen aquestes eines de generació de compiladors, les quals et proporcionen l'algorisme que tracta lèxicament i sintàcticament la gramàtica que defineix el llenguatge font a tractar, i només cal programar l'anàlisi semàntica, la generació de codi i les estructures auxiliars per a tractar la informació, com la taula de símbols. Això significa, que qualsevol canvi que s'hagi d'introduir a la gramàtica, suposarà només un canvi en la definició d'aquesta i no pas en els algorismes que la tracta. No obstant, actualment es poden trobar alguns programadors de compiladors que implementen manualment l'anàlisi lèxica, ja que és la més senzilla i pot ser interessant optimitzar-la d'alguna manera especial (vegeu projecte mono a 7.6 Altres recursos d'interès).

Aquestes eines permeten al programador treballar a un nivell d'abstracció més elevat, concentrant-se només en definir correctament la gramàtica, enlloc de construir l'algorisme recursiu que la reconegui. Normalment aquestes eines generen un compilador o traductor dirigit per la sintaxi, és a dir, la creació d'un *parser* que demana *tokens* a l'analitzador lèxic, i incorpora codi de suport per l'anàlisi semàntica i per a la traducció a codi intermedi.

4.2 Diferents alternatives

Existeix un conjunt d'eines diferents que ajuden a la generació de compiladors. Cada eina es basa en un família de gramàtiques diferent (recordem que hi ha les LL(k) i les LR(k)). S'introduirà

una mica les característiques de cada una de les eines i el seu mode de treball, i finalment s'exposarà quina ha estat l'elecció a l'hora d'implementar el traductor.

4.2.1 Generadors ascendents

Aquestes eines generen *parsers* que construeixen un arbre de derivació acabat per el símbol inicial de la gramàtica, partint dels *tokens* de l'entrada. Són les anomenades gramàtiques LR.

4.2.1.1 LEX i YACC

LEX (*Lexical Analizer Generator*) és una eina generadora de programes pel reconeixement de patrons lèxics dins d'un text (*scanner*). Llegeix un fitxer amb extensió ".l" que conté la descripció de l'analitzador lèxic a generar. Aquesta descripció té la forma de parelles formades per expressions regulars i codi en **C**, anomenades regles lèxiques. El LEX genera com a sortida un fitxer font en **C** que defineix una rutina anomenada *yylex()* que és la que analitza l'entrada en busca d'aparicions que s'adaptin a les expressions regulars. Quan en troba una, executa el codi **C** corresponent.

El YACC (*Yet Another Compiler-Compiler*) és un generador de *parsers* que converteix una descripció d'una gramàtica lliure de context *LALR(1)* en un programa en **C** que reconeix aquesta gramàtica. La gramàtica està formada per una sèrie de regles que estan en BNF (Backus-Naur Form). Aquestes regles expressen l'organització sintàctica dels símbols terminals (donats per l'analitzador lèxic, és a dir, el tipus dels *tokens*), i no terminals (que són construïts per agrupar petites construccions gramaticals, és a dir, cada no terminal té una regla associada que ens mostra els elements dels quals està format). La semàntica es pot realitzar a partir de les accions incorporades a les regles. Cal dir que el codi del *parser* generat pel YACC conté una funció anomenada *yyparse()* que implementa la gramàtica.

El LEX i el YACC interactuen de la següent manera:

La rutina *yyparse()* per tal de llegir els *tokens* fa crides al LEX a partir de la rutina *yylex()* que retorna els tipus de *token* i el valor relacionat a aquest *token*. Per exemple, si detecta 28.03 ha de tornar una constant entera que representarà el tipus de *token*, en aquest cas REAL, que estarà definit en una instrucció *#define*, i el propi valor llegit: 28.03. D'aquesta manera el YACC podrà saber si el

token localitzat és sintàcticament correcte, i podrà realitzar operacions semàntiques a partir del valor.

4.2.1.2 FLEX i BISON

FLEX (A fast scanner generator) és l'eina per generar l'analitzador lèxic i BISON (The YACC-compatible Parser Generator) és l'eina per generar l'analitzador sintàctic. Les diferències entre LEX/YACC i FLEX/BISON són mínimes, però la primera parella està caient en desús a favor d'aquesta última, que està creada més recentment i funciona sota l'entorn GNU. Les diferències principals són les següents:

FLEX

Té les avantatges d'incorporar reconeixement de caràcters de vuit bits, compressió de taules, entre d'altres. En general és compatible amb els fitxers ".l" escrits amb LEX, tot i que presenta petites incompatibilitats:

1. No suporta la variable interna del *scanner yylineno*, que serveix per gestionar el número de línia on ens trobem.
2. No es pot redefinir la rutina *input()* que permet la lectura de caràcters de l'entrada estàndard.
3. No suporta la funció *output()* per escriure un caràcter a la sortida estàndard.
4. El LEX no reconeix l'expressió regular de final de fitxer (EOF).

BISON

Tota gramàtica escrita en YACC pot funcionar en el BISON sense realitzar cap canvi. Per tant, tothom que estigui familiaritzat amb el YACC no tindrà gaires problemes per treballar amb el BISON.

4.2.2 Generadors descendents

Generen *parsers* que partint del símbol inicial de la gramàtica, i aplicant una sèrie de

derivacions, arriben al conjunt de *tokens* de l'entrada. Són les anomenades gramàtiques LL.

4.2.2.1 PCCTS 1.33

PCCTS (Purdue Compiler Construction Set*)* és una eina dissenyada per facilitar la implementació de compiladors i d'altres sistemes de traducció.

PCCTS consisteix en dos processos independents: ANTLR (Another Tool for Language Recognition) i DLG (DFA-based Lexical analyzer Generator que generen analitzadors sintàctics, *parsers*, i analitzadors lèxics respectivament.

ANTLR accepta com entrada un fitxer amb una descripció completa del llenguatge: regles que formaran part del analitzador sintàctic, definició dels components lèxics per l'analitzador lèxic,... i genera una sèrie de fitxers: un programa en **C** que reconeix sentències en aquest llenguatge, un fitxer anomenat "*parser.dlg"* que és una especificació de l'analitzador lèxic que posteriorment serà traduït al **C** per el DLG, un fitxer de tractament d'errors sintàctics, etc.

DLG crea un autòmat que converteix la cadena de caràcters d'entrada en *tokens* que són utilitzats per l'analitzador sintàctic generat per el ANTLR.

Un cop s'han creat tots els fitxers es compilen i s'enllacen els fitxers objecte, així obtindrem el *parser* executable final.

4.2.2.2 ANTLR 2.71

"The ANTLR Translator generator" és una versió completament redissenyada del PCCTS implementada en **Java** que genera codi en: **Java**, **C++** i **Sather**. Es pot dir que el PCCTS va ser la primera generació de la eina programada per Terence Parr, i el ANTLR és la segona generació (vegeu la història de l'eina a 7.6 Altres recursos d'interès). Està programat totalment orientat a objectes, i la manipulació d'errors es fa mitjançant d'excepcions i incorpora moltes novetats i potència a l'hora de tractar la gramàtica.

Com en el PCCTS, la definició completa de la gramàtica està en tot un fitxer i amb l'eina *antlr.Tool* es genera els fitxers amb el codi del *parser* que després han d'ésser compilats amb el llenguatge escollit a l'hora de generar el codi (recordem: **Java**, **C++** o **Sather**).

4.2.3 Conclusió

LEX/YACC i FLEX/BISON són eines molt semblants i accepten gramàtiques LALR(1). Permeten intercalar codi entre les regles, però no permeten el pas de paràmetres. Al tenir separades les especificacions lèxiques de les sintàctiques es produeix un problema de comunicació. Les regles sintàctiques segueixen una notació BNF. El PCCTS accepta gramàtiques LL per tal de construir *parsers* descendents recursius. Rep moltes influències del FLEX/BISON, i segueix una notació EBNF (extended BNF) per les regles. Permet intercalar codi entre les regles i no presenta problemes pel pas de paràmetres. Presenta un tractament d'errors automàtic, o controlat a través d'excepcions. Finalment, el ANTLR té tots els avantatges del PCCTS, a més del suport natiu del **Java** i la plena orientació a objectes.

L'eina principal escollida per a les fases d'anàlisi i de generació de codi és el ANLR 2.71. L'elecció d'aquesta eina ha estat per les següents raons:

1. S'ha disposat de nombrosos exemples per tal d'entendre el seu funcionament.
2. És una eina molt potent, ja que permet que les regles passin paràmetres i retornin valors. Té una recuperació d'errors i un sistema de manipulació d'excepcions que permeten tractar els errors de manera elegant.
3. Perquè permet implementar perfectament les quatre fases necessàries per la realització del cas pràctic: Anàlisi lèxica, anàlisi sintàctica, anàlisi semàntica i generació de codi **Java**.
4. Actualment és utilitzat per una gran base d'usuaris en detriment del PCCTS 1.33, tot i que encara es conserva una petita base que el continua utilitzant.

Ja que el nostre traductor genera codi **Java**, la construcció del traductor amb l'eina ANTLR també serà en codi **Java**, així sempre emprarem aquest llenguatge, que a més és el natiu per aquesta eina. Això també suposa l'avantatge inherent al **Java**, ens estalvia l'elecció d'un sistema operatiu, ja que el **Java** és multiplataforma, i pot ésser executat en qualsevol maquinari que tingui la màquina

virtual de **Java** instal·lada (*Java Runtime* o *JRE*).

4.3 Instal·lació de l'entorn de treball

El traductor necessita les següents dependències:

- Per executar el CPC i generar els fitxers **Java**:

  Màquina virtual de Java 2 instal·lada (JRE 1.2 o superior).

- Per executar el CPC, generar i compilar els fitxers **Java**:

  Java Development Kit 1.2 o superior (JDK 1.2 o superior).

És interessant considerar satisfer la segona opció, ja que amb la primera els fitxers generats per el traductor no es podran compilar, per la qual cosa la utilitat d'aquest queda molt reduïda.

Per instal·lar el *antlr* només cal descomprimir el fitxer que conté el programa en un directori del disc dur creat expressament per contenir-lo. Per a poder fer ús de les classes que incorpora només cal afegir la ruta a les variables d'entorn PATH i CLASSPATH del sistema.

Un cop fet això, ja es té el sistema preparat per poder utilitzar l'eina *antlr*.

5. Construcció del traductor

5.1 Anàlisi lèxica: classe CpcLexer

L'objectiu d'aquesta fase és agrupar la seqüència de caràcters del fitxer font (llegit d'esquerra a dreta) en *tokens* o components lèxics. Un *token* és una seqüència de caràcters determinada.

Els *tokens* els hem agrupat en diferents grups: paraules reservades, identificadors, literals, separadors, operadors i *tokens* a ignorar.

5.1.1 Paraules reservades

Les paraules reservades es defineixen en el *antlr* en el bloc *token*, tal com el següent exemple:

```
tokens
{
   ALGORISME = "algorisme" ;
   FALGORISME = "falgorisme" ;
   CONST = "const"        ;
   FCONST = "fconst" ;
   VAR = "var" ;
   FVAR = "fvar" ;
}
```

Quan el *antlr* realitza la traducció de la nostra gramàtica al fitxer en **Java** que conté la classe que implementa l'analitzador lèxic, construeix el que s'anomena *taula de literals.* Aquesta és una taula *hash* en què la clau és el *token* (**algorisme**,...) i la informació associada és una constant entera que representa el tipus de *token*. Aquestes constants enteres estan declarades en la interfície que genera el *antlr*: “PseudocodiTokenTypes.java”. La classe encarregada de realitzar l'anàlisi lèxica (*CpcLexer*), implementa aquesta interfície, per tal de tenir accés a aquestes constants. En el bloc de *tokens* s'han declarat també un seguit de constants sense cap expressió regular associada. Això s'ha fet per tal de generar constants enteres que siguessin declarades en la interfície “PseudocodiTokenTypes.java” i que han sigut utilitzades més endavant.

5.1.2 Identificadors

Els identificadors són utilitzats per donar noms a les variables, mètodes, etc. L'expressió regular d'un identificador permet que aquest pugui ésser definit com una combinació de lletres i números, amb la restricció que el primer caràcter ha de ser o bé una lletra, o bé el caràcter guió baix ("_"). La definició lèxica de l'identificador seria la següent:

```
IDENTIFICADOR
      :
      ('a'..'z' | '_')
      ('a'..'z' | '_' | '0'..'9')*
      ;
```

Com que el pseudocodi no diferencia majúscules i minúscules, no cal explicitar una combinació de lletres majúscules i minúscules, només posant les minúscules n'hi ha prou.

5.1.3 Literals

Els literals són aquells valors constants escrits en el codi. Els literals admesos són: caràcter, cadena, número enter i número real. Cal notar que *cert* i *fals* també són semànticament literals, però lèxicament són paraules reservades.

La definició de caràcter seria qualsevol caràcter ASCII passat entre cometes simples. Però aquesta definició seria incompleta, per què les seqüències d'escapament també s'inclouen com literals de tipus caràcter. Per tant, el literal caràcter esdevindria de una cometa simple, opcionalment una barra invertida ("\") que indicaria una seqüència d'escapament, un caràcter qualsevol i una altra cometa simple. En expressió regular per el *antlr* restaria:

```
LITERAL_CARACTER:
      '\'' ( ('\\')? (~'\'') )? '\''
      ;
```

El literal cadena és qualsevol combinació de caràcters ASCII entre cometes dobles. Com ja s'ha comentat, aquest literal està inclòs per a simplificar la sortida de missatges, però no té cap tipus de variable associat. La seva definició és:

```
LITERAL_CADENA:
      '\"' (~'\"')* '\"'
      ;
```

Finalment, el literals número enter i número real s'han hagut d'implementar en una sola regla lèxica, ja que sinó produïa un indeterminisme lèxic. El que s'ha fet és que si es troba el punt decimal seguit de dígits, especificarem que el *token* trobat és de tipus real (número decimal), altrament és un número enter, i assignem aquest tipus al *token*. La implementació de la regla és la següent:

```
NUMERO:
    ('0'..'9')+
    ((PUNT('0'..'9')+){_ttype=NUMERO_REAL;}|/*res*/{_ttype=NUMERO_ENTER;})
     ;
```

5.1.4 Separadors i operadors

Els separadors i operadors són definits de manera molt simple, ja que es tracta d'associar un o un parell de caràcters a un *token* específic que després serà tractat per l'analitzador sintàctic. Exemples d'alguns separadors/operadors:

```
PAREN_ESQ       : '('     ;
PAREN_DRET      : ')'     ;
OP_SUMA         : '+'     ;
OP_RESTA        : '-'     ;
```

Cal afegir, com ja s'ha comentat en el capítol 3, que les instruccions estan separades per retorns de carro (o per punt i coma si són en la mateixa línia). Per tant, en l'apartat de separadors també ens apareix el salt de línia, que té la següent definició:

```
SALT_LINIA: ( '\n' {newline();}
             |'\r')
      ;
```

Les claus serveixen per introduir codi que serà introduït directament en el fitxer **Java** generat per l'*antlr*. La instrucció **newline()** incrementa en una unitat la variable interna que porta el comptatge de la línia actual en la que s'està tractant el fitxer d'entrada.

5.1.5 *Tokens* a ignorar

Hi ha uns *tokens* especials que es poden ignorar, i per tant, no seran tractats per l'anàlisi sintàctica. Aquests, un cop han sigut detectats, se li especifica que el seu tipus és: **Token.SKIP**. Aquests *tokens* són els espais en blanc, tabulacions i comentaris. Excepcionalment, els comentaris d'una sola línia (els que comencen per "//") no els ignorarem, per un fet que veurem més endavant en l'anàlisi sintàctica.

La definició de comentari simple és:

```
COMENTARI1:
      "//" (~('\n'|'\r'))* ('\n'|'\r'('\n')?)
      { newline(); }
      ;
```

Si sorgeix un error durant aquesta classe, es llança una excepció que sigui subclasse de **TokenStreamException**. Normalment, es llança **TokenStreamRecognitionException**. El tractament dels errors en la classe lèxica és el del mètode pànic, i per tant un cop llançada l'excepció, el procés de compilació es para.

5.2 Anàlisi sintàctica: classe CpcParser

Una vegada s'ha especificat l'estructura lèxica del llenguatge, cal definir l'estructura sintàctica, que no és res més que la manera com els *tokens* d'entrada es poden combinar per formar sentències permeses pel llenguatge. L'anàlisi sintàctica haurà de seguir estrictament l'especificació del pseudocodi feta en el capítol 3.

Així, per exemple, una classe sintàcticament estaria definida com:

```
classe:
        CLASSE id:IDENTIFICADOR (HERETA hId:IDENTIFICADOR)? (sepa_inst)
        bloc_classe
        FCLASSE (sepa_inst | EOF)
      ;
```

Els *tokens* definits en l'anàlisi lèxica són escrits en majúscules i correspondrien al termes

terminals de la definició de la gramàtica, mentre que els termes escrits en minúscules corresponen a termes no-terminals, és a dir, a altres regles gramaticals.

Quan fem ús d'un terme lèxic en una regla, podem crear una variable que el referenciï. Aquesta es crea posant un nom o identificador seguit de dos punts i el terme lèxic a tractar, com per exemple:

```
CLASSE id:IDENTIFICADOR
```

La variable **id** serà declarada de tipus **Token**, i es pot accedir al text tractat del fitxer font amb el mètode **getText()**. La classe **Token** conté la informació del tipus i del text de la unitat lèxica reconeguda. A més, també desa el número de línia en el qual es troba el *token*, la columna i altra informació.

L'interrogant al final del terme **hereta** significa que és opcional, i és equivalent a definir la regla de la forma o bé **hereta** o bé res:

```
(HERETA hId:IDENTIFICADOR)?  <==> ((HERETA hId:IDENTIFICADOR) | /*res*/)
```

Les instruccions van separades per un retorn de carro, el qual està definit en la regla **sepa_inst**, que és la separadora d'instruccions:

```
sepa_inst: ( SALT_LINIA | COMENTARI1)+
      ;
```

El **SALT_LINIA** ja s'ha vist anteriorment com està declarat. El **COMENTARI1** és el comentari d'una sola línia, que comença per "//" i s'acaba a final de línia. Aquest comentari no ha pogut ésser ignorat en l'anàlisi lèxica, ja que d'altra forma el *token* estaria format per el comentari més el retorn de carro corresponent, necessari per la pròpia definició de comentari. Això faria que no es pogués reconèixer la instrucció, ja que no es trobaria el retorn de carro.

Per exemple el següent pseudocodi:

```
classe Primera // Prova d'una classe
fclasse
```

Si el *token* comentari no s'hagués inclòs en l'anàlisi sintàctica i s'hagués ignorat, veiem com no coincideixen l'arbre de derivació amb l'arbre gramatical definit:

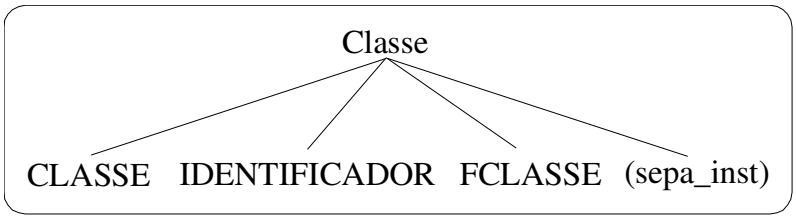


Figura 5.1 Arbre de derivació del pseudocodi d'exemple

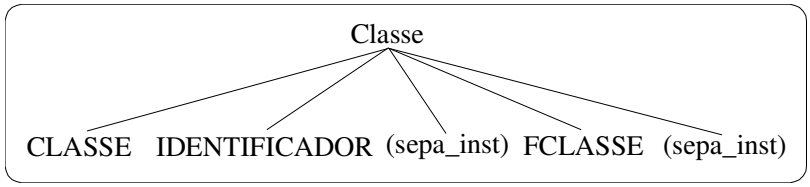


Figura 5.2 Arbre gramatical de la regla

Per tant, no s'acceptaria el fitxer i generaria error de compilació, encara que podem comprovar que és del tot correcte. Degut a que el comentari acaba en retorn de carro, l'analitzador lèxic afegeix el retorn de carro al *token* **COMENTARI** i per tant, no es detecta el *token* **SALT_LINIA**. Per això hem optat la solució de no ignorar-lo i definir el separador d'instruccions com un salt de línia o un comentari de línia simple (que porta implícitament el salt de línia).

Un predicat difícil de definir per a la gramàtica ha sigut el de variable. Una variable pot anar precedida de **jo** o bé **super**, per això s'ha definit el predicat *variable* i *post_variable*. El primer defineix la possible precedència anterior, mentre que el segon defineix estrictament la variable.

Vegem la definició:

```
variable:
      JO (PUNT post_variable)?
      | SUPER ( PUNT post_variable | crida_funcio )
      | post_variable
      ;
```

Llavors, *post_variable* s'encarrega de definir una variable que sigui un identificador sol, un accés a un membre d'una classe, un accés a un element d'una taula o una crida a una funció. La definició és la següent:

```
post_variable:
      (id:IDENTIFICADOR
      ( PUNT post_variable
      |(index_taula)(PUNT post_variable | /* res */)
```

```
        | crida_funcio (PUNT factInf1=post_variable[factInf] | /* res */)
        | /* res */)
        )
```

El *parser* LL ens produïa un indeterminisme entre la instrucció assignació i la de crida a una funció, ja que les dues comencen amb una **IDENTIFICADOR**. Això s'ha solucionat fent un predicat mixt que agrupés tant l'assignació, com la crida a una funció. Concretament, el predicat és el següent:

```
inst_assignacio_crida_funcio:
      variable
      (OP_ASSIGNACIO expressio
      | /* RES: En aquest cas es una crida */
      )
      ;
```

Aquesta definició ens produeix el problema que sintàcticament són correctes certs predicats, que realment són invàlids, com per exemple una instrucció o només hi hagi una variable. Com per exemple:

```
var1
```

Per solucionar-ho, ho hem tractat en l'anàlisi semàntica, de manera que si no és una assignació, comprovar que realment la instrucció és una crida a una funció i no pas una variable.

En la definició de les regles sintàctiques es poden especificar paràmetres, com si fossin funcions (en realitat, quan l'*antlr* crea el *parser* tradueix les regles sintàctiques en funcions). També es pot especificar que retornin un valor. La manera de fer-ho és:

```
nom_regla[paràmetres] returns[tipus identificador]
```

(Els corxets són caràcters imprescindibles).

Aquests paràmetres són utilitzats posteriorment per l'anàlisi semàntica, per tal de poder passar informació entre les regles, i realitzar així les comprovacions semàntiques corresponents.

Els errors generats durant la fase sintàctica es tracten amb el mètode de recuperació. Això vol dir que es reporta a l'usuari el missatge d'error, però el procés de compilació continua. Normalment les excepcions generades en aquesta etapa per identificar errors sintàctics són subclasses de

RecognitionError. Normalment les llançades són: **MismatchedTokenException** i **NoViableAltException**.

5.3 Anàlisi semàntica: comprovacions i missatges

L'anàlisi semàntica examina l'arbre de derivació construït per l'anàlisi sintàctica per trobar errors semàntics: tipus incompatibles en una assignació o en una operació, unicitat dels identificadors,... També s'encarrega de convertir tipus, si és possible, abans de generar un error d'incompatibilitat de tipus.

A continuació es detallaran els aspectes que controla l'analitzador semàntic, les excepcions que es generen i els respectius missatges d'error (en cursiva són substituïts per el seu corresponent valor a l'hora de mostrar el missatge d'error).

5.3.1 Errors en la declaració de variables

1. Comprovar que l'identificador de la nova variable no estigui ja declarat.

   Es llança l'excepció **IdAlreadyDeclaredException** i el missatge d'error serà:

   "*fitxer*:*línia*: '*identificador*' ja declarat"
2. Quan la variable que es declara és una referència a una classe i no es té carregada la interfície d'aquesta en la taula de símbols, s'intenta llegir el fitxer per carregar la interfície, si el fitxer no existeix, es llança l'excepció **CantLoadClassFileException** i es genera el següent missatge:

   "fitxer:*línia*: no es troba la classe '*identificador*'*"*

5.3.2 Errors en la declaració de constants

1. Comprovar que l'identificador de la nova constant no estigui ja declarat.

   Es llança l'excepció **IdAlreadyDeclaredException** i el missatge d'error serà:

   "*fitxer*:*línia*: '*identificador*' ja declarat"
2. Comprovar que la constant és de tipus bàsic o elemental (enter, caràcter, real o booleà).

   Es llança l'excepció **NeedElementalTypeException** i el missatge d'error serà:

"*fitxer*:*línia*: la constant '*identificador*' no es de tipus valid"

3. Comprovar que el tipus de la constant coincideix amb el tipus del valor assignat.

   Hi ha dos possibles missatges d'error:

   - Si el traductor es capaç de detectar quin són els dos tipus:

   "*fitxer*:*línia*: tipus incompatibles. Es necessita *'tipus*' trobat *'tipus*'"

   - Si el traductor no es capaç de saber algun dels dos tipus (ex. Una variable no declarada):

   "*fitxer*:*línia*: tipus incompatibles"

   En els dos casos, l'excepció llançada és **TypeInvalidException**.

5.3.3 Errors en la declaració de mètodes/funcions

1. Comprovar que l'identificador del nou mètode no estigui ja declarat, en cas que ho estigui, comprovar que la seva signatura sigui diferent (sobrecàrrega de mètodes). Si coincideixen en l'assignatura es genera l'excepció **IdAlreadyDeclaredException** i el seu missatge serà:

   "*fitxer*:*línia*: '*identificador*(*signatura*)' ja declarat"

2. Comprovar que el constructor té el mateix nom que la classe.

   Es llança l'excepció **ConstructorNotValidException** i el missatge d'error serà:

   "*fitxer*:*línia*: el constructor no te el mateix nom que la classe"

3. Comprovar que els membres de la classe no tenen el mateix nom que la classe (només té el mateix nom el constructor).

   Es llança l'excepció **FuncIdNotValidException** i el missatge d'error serà:

   "*fitxer*:*línia*: el membre no pot tenir el mateix nom que la classe"

5.3.4 Errors en les sentències

1. Comprovar que la variable que s'utilitzi estigui declarada.

   Es llança l'excepció **IdNotDeclaredException** i el missatge d'error serà:

   "*fitxer*:*línia*: '*identificador*' no declarat"

2. Comprovar que el tipus de la part esquerra i dreta d'una assignació concordin.

   Hi ha dos possibles missatges d'error:

   - Si el traductor es capaç de detectar quin són els dos tipus:

"*fitxer*:*línia*: tipus incompatibles. Es necessita '*tipus*' trobat '*tipus*'"

- Si el traductor no es capaç de saber algun dels dos tipus (ex. Una variable no declarada):

"*fitxer*:*línia*: tipus incompatibles"

En els dos casos, l'excepció llançada és **TypeInvalidException**.

3. Comprovar que la variable utilitzada com a comptador en una instrucció **per** sigui de tipus enter, així com la instrucció d'inici, de fi i de **pas**.

   Els missatges d'error corresponents seran:

   - "*fitxer*:*línia*: es necessita un enter per la variable del per"
   - "*fitxer*:*línia*: es necessita un enter per l'expressio d'inici del per"
   - "*fitxer*:*línia*: es necessita un enter per l'expressio de fi del per"
   - "*fitxer*:*línia*: es necessita un enter per l'expressio del pas"

   En tots els casos, l'excepció llançada és **RequiredEnterException**.

4. Comprovar que la condició d'una instrucció **si** és de tipus booleà.

   Es llança l'excepció **RequiredBooleaException** i el missatge d'error serà:

   "*fitxer*:*línia*: es necessita un boolea per la condicio del si"

5. Comprovar que la condició d'una instrucció **cas** és de tipus booleà.

   Es llança l'excepció **RequiredBooleaException** i el missatge d'error serà:

   "*fitxer*:*línia*: es necessita un boolea per la condicio del cas"

6. Comprovar que la condició d'una instrucció **mentre** és de tipus booleà.

   Es llança l'excepció **RequiredBooleaException** i el missatge d'error serà:

   "*fitxer*:*línia*: es necessita un boolea per la condicio del mentre"

7. Comprovar en les assignacions que l'expressió de l'esquerre sigui una variable i d'igual manera en el **per** la variable que fa de comptador.

   Es llança l'excepció **RequiredVariableException** i el missatge d'error serà:

   "*fitxer*:*línia*: '*identificador*' es un valor. Es necessita una variable"

8. Comprovar en les assignacions que l'expressió de l'esquerre no és una constant.

   Es llança l'excepció **CantModifyConstantException** i el missatge d'error serà:

   "*fitxer*:*línia*: no es pot modificar la constant *'identificador*'"

9. Línies que continguin només una expressió, o una variable no són instruccions correctes (per exemple línies que només continguin 'var1').

   Es llança l'excepció **NotInstructionException** i el missatge d'error serà:

"*fitxer*:*línia*: *'text'* no es una instruccio correcte"

10. Cal comprovar que les dimensions de la taula a l'hora d'accedir-hi siguin les correctes conforme la declaració d'aquesta.

    Es llança l'excepció **BadTableDimensionException** i el missatge d'error serà:

    "*fitxer*:*línia*: les dimensions de la taula '*identificador*' no son correctes"

11. La instrucció **llegir** només pot ser aplicada sobre variables de tipus bàsic o elemental, però no sobre classes.

    Es llança l'excepció **CantReadVariableException** i el missatge d'error serà:

    "*fitxer*:*línia*: '*identificador*' no es un tipus basic, no es pot aplicar 'llegir'"

5.3.5 Errors en les expressions

1. Comprovar que una variable que sigui utilitzada com una taula, estigui declarada com a taula.

   Es llança l'excepció **RequiredTableException** i el missatge d'error serà:

   "*fitxer*:*línia*: variable *'identificador'* no es una taula"

2. Comprovar que els valors que es fan servir per indexar la taula (ex: taula1[2][4]) siguin de tipus enter.

   Es llança l'excepció **RequiredEnterForIndexException** i el missatge d'error serà:

   "*fitxer*:*línia*: es necessita un enter per indexar la taula *'identificador'*"

3. Comprovar que els operands dels operadors **div** i **mod** són enters.

   Els missatges d'error corresponents seran:

   - "*fitxer*:*línia*: es necessita un enter per els operadors del mod"
   - "*fitxer*:*línia*: es necessita un enter per els operadors del div"

   En els dos casos, l'excepció llançada és **RequiredEnterException**.

5.3.6 Errors en l'accés a les classes

1. Comprovar quan s'accedeix a un membre d'una classe que sigui públic, que no tingui accés privat.

   Es llança l'excepció **PrivateAccesException** i el missatge d'error serà:

   "*fitxer*:*línia*: '*identificador*' te acces privat"

2. Quan s'accedeix a un membre de classe, comprovar que s'hi ha posat el nom de la classe en qüestió, i no el d'una instància de la classe.

   Es llança l'excepció **ClassMemberAccesException** i el missatge d'error serà:

   "*fitxer*:*línia*: '*identificador*' es un membre de classe"

3. Quan s'accedeix a un membre posant el nom d'una classe enlloc del d'una instància, comprovar que al membre al qual s'accedeix és un membre de classe.

   Es llança l'excepció **NotMemberClassException** i el missatge d'error serà:

   "*fitxer*:*línia*: '*identificador*' no es un membre de classe"

4. Dintre un membre de classe, només es poden accedir a les seves variables locals i a altres membres de la classe, però no als membres d'instància.

   Es llança l'excepció **CantRefenceInstanciaException** i el missatge d'error serà:

   "*fitxer*:*línia*: '*identificador*' no pot ser referenciat en un membre de classe"

5. Quan es declara una instància d'una classe, o s'accedeix a aquesta i no es té carregada la interfície en la taula de símbols, s'intenta llegir el fitxer per carregar la interfície, si el fitxer no existeix, es llança l'excepció **CantLoadClassFileException** i es genera el següent missatge:

   "*fitxer*:*línia*: no es troba la classe '*identificador*'"

6. Quan es crida a un mètode, s'ha de comprovar que estigui declarat. Si hi ha més d'un declarat amb el mateix nom s'ha de comprovar la signatura. Si no coincideix, el mètode no existeix i es llança l'excepció **MetodeNotExistException** generant el següent missatge:

   "*fitxer*:*línia*: no es troba el metode '*identificador*(*signatura*)'"

7. Comprovar que la crida **super()** és la primera instrucció en un constructor.

   Es llança l'excepció **SuperFirstStatementException** i el missatge d'error serà:

   "*fitxer*:*línia*: la crida super ha de ser la primera instruccio en un constructor"

8. Comprovar que la crida **super()** és crida en un constructor.

   Es llança l'excepció **CallConstSuperException** i el missatge d'error serà:

   "*fitxer*:*línia*: nomes es pot cridar constructor de la superclasse en un constructor"

9. Comprovar que els mètodes declarats en la interfície són implementats dintre el bloc d'implementació.

   Es llança l'excepció **MetodeNotImplementedException** i el missatge d'error serà:

   "*fitxer*:*línia*: metode '*identificador*(*signatura*)' no implementat en la classe"

10. Comprovar que els mètodes implementats estan declarats prèviament en el bloc d'interfície.

    Es llança l'excepció **MetodeNotDeclInterfException** i el missatge d'error serà:

"*fitxer*:*línia*: metode '*identificador*(*signatura*)' no declarat en interficie"

11.Comprovar que els mètodes s'implementen només una vegada.

Es llança l'excepció **MetodeAlreadyImplementedException** i el missatge d'error serà:

"*fitxer*:*línia*: metode '*identificador*(*signatura*)' ja esta implementat"

5.4 Tractament d'errors

En qualsevol de les fases es poden trobar errors. Després de detectar-ne un, s'ha de tractar d'alguna manera per poder continuar la compilació. Com es va veure en el capítol 2, hi ha diferents maneres d'actuar davant d'un error en el codi font, depenent de la fase de compilació en el qual es troben:

1. **Anàlisi lèxica**. Actuem amb el mètode pànic: s'abandona la compilació al trobar un error.
2. **Anàlisi sintàctica**. Mètode de recuperació: s'intenta, llegint més *tokens*, que el procés de compilació es restauri.
3. **Anàlisi semàntica**. Mètode de restauració "intel·ligent": es realitzen canvis semàntics, de manera intel·ligent, per continuar el procés de compilació. Per exemple, si no coincideixen els tipus de les variables, pot canviar el seu tipus per tal de d'arreglar l'error.

Per tal de tractar els errors, s'ha utilitzat excepcions. El propi *antlr* produeix un tractament per els errors lèxics i sintàctics, conseqüentment ens hem centrat majoritàriament en els errors semàntics. Per els errors semàntics, s'ha definit la classe **PSemanticException**, que és la classe pare de totes les excepcions semàntiques. Quan falla alguna comprovació semàntica, es llança l'excepció corresponent amb la instrucció *throw* i quan es captura amb el bloc *catch* es crida a *reportError*. Aquesta funció és l'encarregada de reportar tots els errors, ja siguin lèxics, sintàctics com semàntics. Aquesta centralització permet portar un comptatge de tots els errors trobats en el fitxer tractat i mostrar-ho al finalitzar el tractament del fitxer.

Les excepcions creades per el traductor són subclasses de **PSemanticException**, i aquesta alhora és subclasse de **ANTLRException**. Aquestes noves excepcions no s'han derivat de **SemanticException**, ja que aquesta està destinada a ser llançada quan falla uns predicats semàntics,

incorporats en l'anàlisi sintàctica que proporciona el *antlr*. A més, interessava que les excepcions semàntiques incorporades tinguessin un pare diferent a la dels errors sintàctics (**RecognitionError**), ja que així les podem tractar a part.

Quan en una expressió s'opera enters amb reals, i s'ha d'assignar el resultat a un enter, el traductor intenta canviar el tipus del real a enter. Això provoca un *warning* o avís, ja que es pot perdre precisió en fer el canvi. Aquest avís és generar amb l'excepció **LossPrecisionWarningException** que deriva de **PSemanticWarning**. Totes les excepcions que representen avisos han de derivar de **PSemanticWarning**. Tot i que actualment només existeix aquesta, si en un futur s'implementen més avisos, les seves respectives excepcions haurien de ser subclasses **PSemanticWarning**.

A la pàgina següent es pot observar una figura amb la jerarquia d'excepcions utilitzada en el traductor. Les excepcions amb el requadre blanc són les que venen incorporades amb el *antlr*, mentre que les del requadre gris són creades especialment per el traductor. No obstant, les excepcions **MismatchedTokenException** i **NoViableAltException** han sigut retocades lleugerament per tal que el text que mostrin quan es reporta un error estigui en català, enlloc de l'anglès.

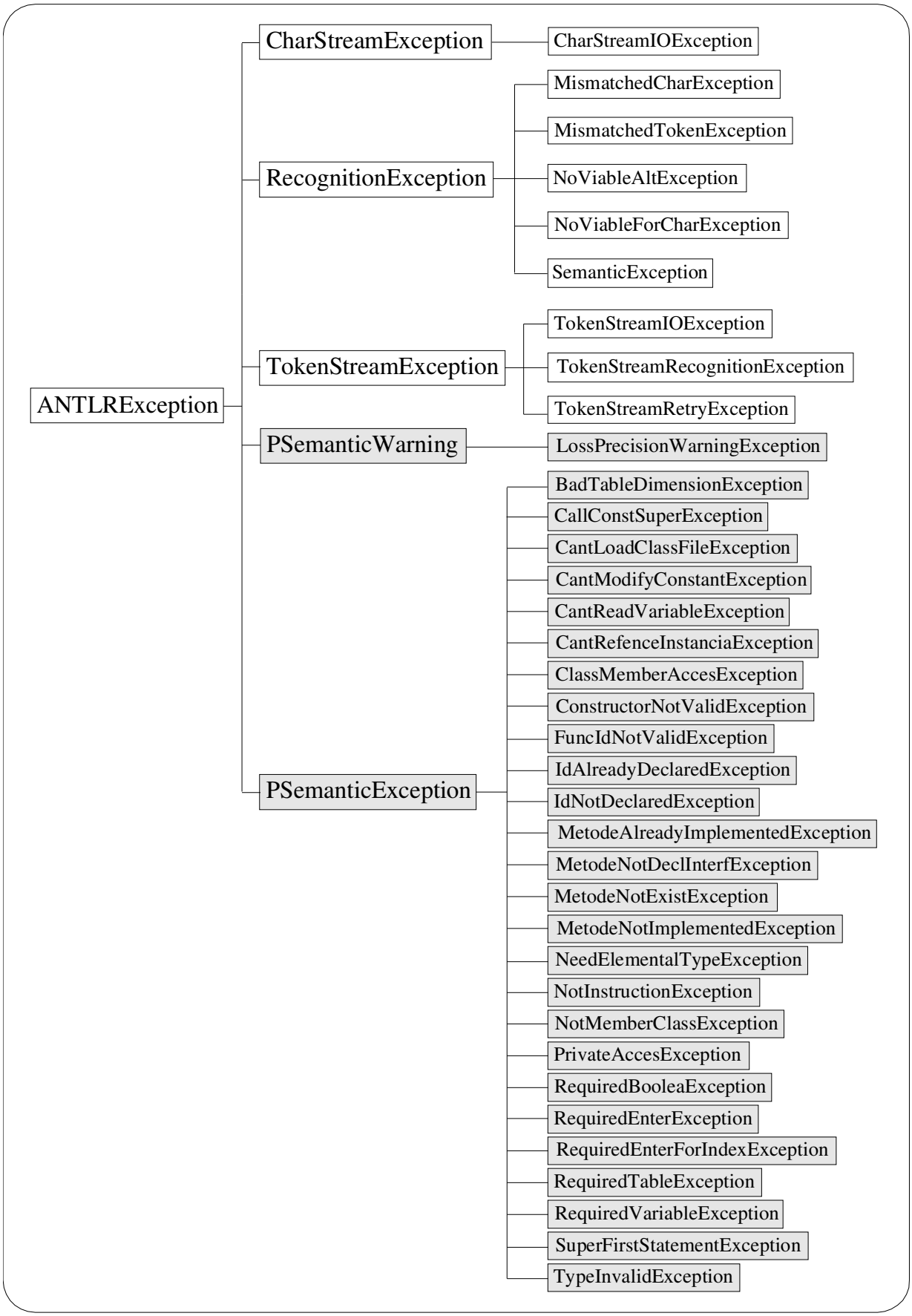


Figura 5.3 Jerarquia d'excepcions del traductor

5.5 La taula de símbols

La taula de símbols serveix per a guardar la informació dels *tokens* i implementar la semàntica del nostre llenguatge (bàsicament comprovació de tipus) i s'implementa utilitzant diverses estructures de dades.

5.5.1 Estructura de dades

L'estructura de dades més senzilla que existeix per una taula de símbols és l'estructura de dades tipus llista, és a dir, una llista lineal de registres que contenen un nom per identificar el registre de manera única, i la informació associada a aquest nom. Aquesta no és gaire utilitzada si no és per compiladors en estat experimental, ja que ofereix un rendiment molt pobre. S'ha de tenir en compte que la taula de símbols serà consultada freqüentment per establir el tipus d'una variable o mètode, la visibilitat, etc. Per tant, ens interessa una estructura de ràpid accés.

El mètode més utilitzat en els compiladors per la seva eficiència i flexibilitat és una taula de dispersió o taula de *hash*. Es sol utilitzar una tècnica de dispersió oberta, ja que no hi ha límit en quan el número d'entrades que pot contenir la taula de símbols. El sistema consisteix en una funció *hash* que rep com entrada la clau de l'entrada (en aquest cas seria el nom de l'identificador) i com a sortida retorna un enter que correspon a un índex de la taula. Per a què la funció de *hash* sigui bona, cada índex de la taula ha de correspondre a un sol identificador. D'aquesta manera, s'accediria molt ràpidament (eficiència O(1)) a una entrada de la taula utilitzant com a clau l'identificador. Però en la realitat això no es sol donar i apareixen les col·lisions, que són dues entrades amb el mateix índex. Per a solucionar això, s'implementa un sistema de cubetes, que és una llista enllaçada en l'entrada que conté les col·lisions. Això fa que l'eficiència no sigui O(1), però sigui molt propera, depenent del nombre de col·lisions que produeixi la funció de dispersió. Per això convé que aquesta sigui bona.

En la pàgina següent es pot veure una figura que mostra com actuaria una taula de símbols implementada amb una estructura de taula de dispersió oberta.

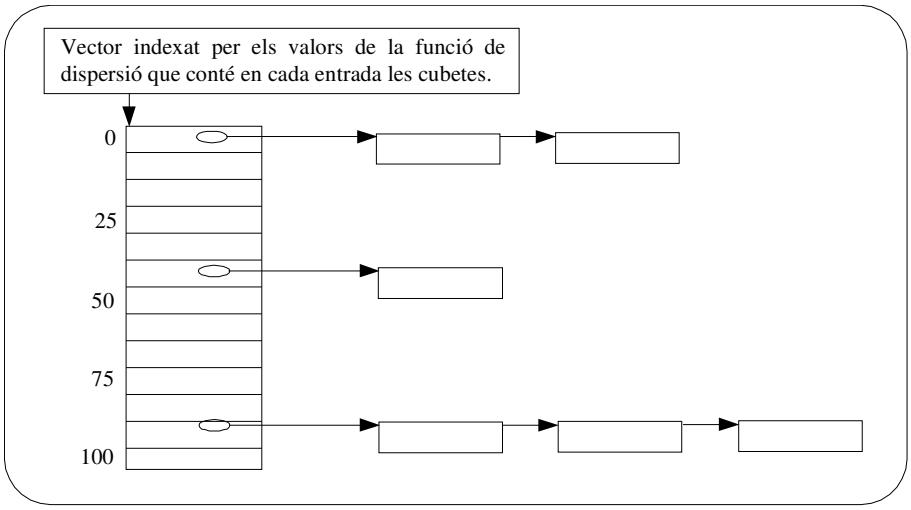


Figura 5.4 Estructura d'una taula de símbols implementada amb una taula de dispersió

Es poden diferenciar dues parts en l'estructura de dades:

1. La taula de dispersió formada per un vector fixa de MAX elements, i que conté les diferents entrades per a cada identificador.
2. Les llistes enllaçades de cada entrada anomenades cubetes. Cada registre en la taula de símbols apareix només en una cubeta.

Existeixen altres estructures per implementar la taula de símbols, com ara les *skiplist*, arbres binaris, arbres binaris balancejats,... Però s'utilitzarà la taula *hash* ja que la proporció el rendiment és prou bo i no cal un codi massa complicat.

5.5.2 Tractament dels àmbits

Cada mètode representa un àmbit, que té les seves variables locals, i fins i tot una classe es pot considerar un àmbit global. Cada classe té el seus àmbits, i la taula de símbols ha d'estar preparada per a tractar-los. Hi ha dues maneres d'implementar els àmbits: amb una pila de taula de símbols o amb una sola taula.

Una pila de taula de símbols fa que tinguem moltes taules de símbols en memòria. Està molt marcada la diferència amb cada àmbit. Quan es busca un identificador, es cerca en l'últim element de la pila. Si no es troba en aquest, s'ha de buscar en l'anterior, i sinó en l'altre, fins arribar al primer

de la pila. D'aquesta manera es soluciona el problema de la visibilitat, ja que les variables declarades en la classe han de ser visibles en els mètodes locals. Aquest model de treball és prou bo, però ens representa un problema. Quan hem de treballar amb una classe externa, hem de carregar la seva taula de símbols per saber quins membres de la classe són visibles (públics), i per tant hi podem accedir. Carregar la interfície pública de la classe pot ser una mica dificultós amb aquest mètode.

Per implementar els àmbits en una sola taula de símbols s'utilitza la pròpia clau o identificador. Es manté una pila o vector de cadenes, separades per algun signe (en el nostre cas estan separades per "::"). Cada cadena representa un àmbit, i l'àmbit actual està definit per la concatenació de totes les cadenes desades en la pila. Quan s'insereix un símbol, el que en realitat s'afegeix a la taula de dispersió és l'àmbit actual concatenat amb el símbol. Quan es cerca un identificador, se li afegeix a aquest l'àmbit actual i prova si existeix l'entrada en la taula *hash*. En cas negatiu, desempilaria un àmbit de l'identificador, que consistiria en esborrar l'última cadena de l'expressió de l'àmbit actual, i tornar a cercar en la taula, fins que es trobi, o es desempilin tots els àmbits. L'algorisme de cerca d'un identificador és el següent:

```
public LinkedList lookupNameInCurrentScope(String name)
{
      String scope = currentScopeAsString();
      String scopedName;
      LinkedList list = null;

      while (list == null && scope != null)
                {
                        scopedName = addScopeToName(scope, name);
                        list = (LinkedList)symTable.get(scopedName);
                        scope = removeOneLevelScope(scope);
      }
      return list;
}
```

La funció **currentScopeAsString** retorna l'àmbit actual (les cadenes concatenades separades per "::"). El mètode **addScopeToName** concatena l'àmbit amb el nom, separant-los amb "::" i **removeOneLevelScope** s'encarrega d'esborrar l'última cadena de l'àmbit passat per paràmetre.

Aquest sistema és molt útil, ja que a l'hora de carregar els símbols d'una classe externa només cal carregar la taula *hash* degudament emplenada.

5.5.3 Informació desada

En cada entrada de la taula *hash* hi ha una llista enllaçada, independentment de la possible llista enllaçada de la cubeta. Això permet desar per el mateix identificador, diferents entrades, i implementar així la sobrecàrrega de mètodes. En l'orientació a objectes, es poden implementar mètodes amb el mateix nom, però diferent signatura, com ja es va veure en el capítol 3. Quan es declara o es crida una funció, primerament es comprova que l'identificador estigui en la taula de símbols (o sigui visible en l'àmbit actual) i després es comproven els paràmetres. En cas que falli la última comprovació, busca el següent de la llista i torna a comparar els paràmetres, i així successivament. Cada element de la llista és un objecte **TNode**, el qual guarda informació sobre els paràmetres (si l'identificador és un mètode), el tipus, la classe, etc.

Les claus de la taula de símbols, el nom de la funció o identificador, s'introdueixen sempre en minúscules, i es fa la cerca dintre els àmbits també amb la cadena passada a minúscules. Així ens estalviem problemes amb les comparacions, ja que el pseudocodi no distingeix majúscules i minúscules. Al treballar amb minúscules a la taula de símbols, fem que aquesta tampoc distingeixi entre majúscules i minúscules, que és el que ens interessa.

A més, en la classe de la taula de símbols hi ha, independentment de la taula de dispersió, una llista amb els mètodes declarats en la classe actual, de tal manera que ens permet saber si el mètode està declarat en la interfície, o si s'ha implementat en la classe, i generar l'error corresponent si no es compleixen les condicions.

5.6 Classes auxiliars: ParamInfo i FuncInfo

Una vegada hem extret la informació que ens interessa de la taula de símbols, aquesta s'ha de propagar per les regles sintàctiques, per tal de fer les comprovacions semàntiques corresponents. Però no cal propagar tota la classe **TNode**, només un subconjunt de la informació. La classe

ParamInfo conté informació sobre el paràmetre o identificador que s'està tractant. Es molt útil per emmagatzemar la informació d'un identificador (tipus, text *parsejat*, ...). A més, per propagar la informació dels paràmetres d'una funció es fa servir una taula de **ParamInfo**, on cada element conté la informació d'un paràmetre.

D'altra banda, la comprovació semàntica que s'assegura que tots els mètodes de la interfície de la classe han estat implementats, ha requerit implementar en la taula de símbols una llista encadenada que conté els mètodes declarats en la interfície de la classe. Aquesta llista està formada per objectes de tipus **FuncInfo**, que contenen informació sobre la signatura de la classe i un booleà per saber si s'ha implementat.

5.7 Generació de codi: classe CpcGenJava

Una vegada acabat l'anàlisi lèxica, sintàctica i semàntica, cal generar el codi **Java** corresponent al programa en pseudocodi tractat. La classe **CpcGenJava** s'encarrega de generar el codi **Java** per a cada instrucció: depenent de la instrucció detectada, el traductor crida a una funció o a una altra de la classe amb els paràmetres necessaris i aquesta escriu en el fitxer corresponent la traducció.

La classe escriu fitxers de text pla amb la traducció al **Java** i aquests fitxers poden ésser compilats directament amb el compilador de *Sun Microsystems* **javac**, o qualsevol altre compilador estàndard de **Java**.

Els membres privats de les classes en pseudocodi són generats com membres **protected** en **Java**. Així aquests membres es comporten com privats per a classes externes, però permet que siguin accessibles per les subclasses que es puguin generar. El pseudocodi no fa aquesta distinció, tots els mètodes són visibles per les seves classes derivades.

La instrucció **cas** de pseudocodi ha estat implementada com un **if-else** en **Java**, enlloc de un **switch**. Això és degut a què el **cas** del pseudocodi permet comparacions complexes de més petit igual, entre d'altres; mentre que el **switch** del **Java** no les permet.

A l'hora de traduir les instruccions d'entrada i sortida **escriure**, **escriureln** i **llegir** s'utilitzen les funcions de la classe **_____InputOutput**, que forma part del traductor. En cada classe creada per el traductor es declara automàticament una variable de tipus **_____InputOutput** amb un nom especial (comença amb molts caràcters de guió baix), per tal que no entri en conflicte amb cap variable creada per l'usuari. Això permet una entrada i sortida molt més acurada. Per exemple, s'ha adaptat el mètode de **llegir** caràcters, de tal manera que si s'executa varis cops seguits, permet simular que llegeix una cadena del teclat (es pot veure en la classe "cadena.pse" dels exemples de pseudocodi del suport CD-ROM). La utilització d'aquesta classe d'entrada i sortida permet tenir més control en aquestes operacions, que no pas si s'hagués introduït tot el codi dintre de la classe traduïda.

La classe **CpcGenJava** implementa la interfície "PseudocodiTokenTypes.java" per tal de saber de quins tipus són els identificador que es passen per paràmetre des del *parser* i poder generar correctament el codi **Java**.

Cal dir que es pot considerar aquesta classe com el *back-end* del traductor. De manera que podríem dissenyar una classe que es digués **CpcGenCplusplus**, que generés codi **C++**, i tenir amb el mateix *front-end* que tracta el pseudocodi, traduccions a dos llenguatges diferents: **C++** i **Java**.

6. Proves d'execució

6.1 Les proves

Després de tot el treball fet per a la construcció del traductor de pseudocodi a **Java**, cal demostrar que tot l'esforç ha donat el seu fruit. En aquest capítol abordarem diversos exemples de codis escrits en pseudocodi i tractats per el traductor, per demostrar la seva potència i versatilitat.

6.2 Exemple 1: números primers

Exemple dels números primers, el típic exemple. Demana un número a l'usuari i contesta si és o no un número primer. Aquest exemple és un algorisme amb errors. Comprovarem que executant el traductor aquest ens senyalarà els errors que hi ha en el codi. El llistat del programa és el següent:

```
algorisme primer
var
      num,divisor,reste: enter
      parar: boolea
fvar

      escriure("Programa de numeros primers\n")
      escriure("Escriu un numero:")
      num <- llegir()

      divisor <- 2
      parar<-fals
      mentre ((divisor < num) i (no parar)) fer
                si (num mod divisor)=0 llavors
                          parar <- 39          // Error de tipus
                altrament
                          divisor <- divisor + 1
                fsi
      fmentre

      si (45) llavors                // Error es necessita un boolea
                escriureln("El numero no es primer")
      altrament
                escriureln("Aquest es un numero primer!")
      fsi

falgorisme
```

Si executem la comanda per a començar el procés de traducció:

```
java cpc primer.pse
```

Obtindrem la següent sortida:

```
[koss@MOBILE simple]$ java cpc  primer.pse
primer.pse:15: tipus incompatibles. Es necessita 'boolea' trobat 'enter'
primer.pse:21: es necessita un boolea per la condicio del si
2 errors
[koss@MOBILE simple]$
```

Imatge 6.1 Captura de la sortida de la traducció de “primer.pse”

La sortida ha generat dos errors. Si passem a mirar el primer veiem que a la següent línia ens marca un error:

```
parar <- 39          // Error de tipus
```

És cert, la variable *parar* està declarada com a booleà. Si ens fixem una mica, descobrim que per a què l'algorisme funcionés, el *39* hauria de ser canviat per *cert*.

El segon error es troba a la línia 21:

```
si (45) llavors                // Error es necessita un boolea
```

Veiem com en aquí també s'ha comportat correctament el traductor, ja que la condició d'avaluació del **si** ha d'ésser un booleana, mentre que en el codi és un enter. De seguida es pot observar que realment el que hi hauria d'anar és la variable *parar*.

6.2 Exemple 2: classe Punt3d

En aquest exemple s'implementaran dos fitxers: una classe anomenada Punt3d i un altre fitxer amb un algorisme per tal de provar l'accés a membres de la classe. Tornarem a provar si el traductor troba els errors introduïts en l'escriptura del codi.

Aquí podem veure el codi de la classe “Punt3d.pse”:

```
classe Punt3d
/* Interficie: Declaracio de membres publics */
interficie
      atributs
                var
                          x,y,z: enter
                fvar
      fatributs
      metodes
                constructor Punt3d()
                metode sumar(p1: Punt3d,p2: Punt3d) retorna pf: Punt3d
      fmetodes
finterficie
/* Implementacio: Implementa tant els membres publics com els privats */
implementacio
      constructor Punt3d()
                x <- 0; y <- 0; z <- 0
      fconstructor

      /*Suma els dos numeros passats per parametre i retorna el resultat*/
      funcio sumar_simple(n1: enter,n2: enter) retorna r: enter
                r <- n1 + n2
                retorna r
      ffuncio
      /* Suma amb un Punt3d, component a component */
      metode sumar(p1: Punt3d,p2: Punt3d) retorna pf: Punt3d
                pf.x <- sumar_simple(p1.x,p2.x)
                pf.y <- sumar_simple(p1.y,p2.y)
                pf.z <- sumar_simple(p1.z,p2.z)
                retorna pf
      fmetode

fimplementacio

fclasse
```

A continuació el codi de l'algorisme “MainPunt3d.pse”:

```
/****************************************
 * Algorisme de proves de la classe Punt3d
 *
 ****************************************/
algorisme MainPunt3d
      var
                p1,p2: Punt3d
                n1,n2,n3: enter
      fvar

      n1 <- crear Punt3d()
      p1 <- crear Punt3d()
      p2 <- crear Punt3d()
      n1 <- 3; n2 <- 4; n3 <- 1

      p1.sumar(p1,p2)
      p1.sumar_simple(n1,23)
      Punt3d.sumar(p1,p2)

falgorisme
```

Si executem la comanda per a començar el procés de traducció:

```
java cpc MainPunt3d.pse
```

Obtindrem la següent sortida:

```
[koss@MOBILE Punt3d]$ java cpc MainPunt3d.pse
MainPunt3d.pse:11: tipus incompatibles. Es necessita 'enter' trobat 'Punt3d'
MainPunt3d.pse:17: 'sumar_simple' te acces privat
MainPunt3d.pse:18: 'sumar' no es un membre de classe
3 errors
[koss@MOBILE Punt3d]$
```

Imatge 6.2 Captura de la sortida de la traducció de “MainPunt3d.pse”

La sortida ha generat 3 errors. Comprovarem que són certs. El primer fa referència a la línia 11 del fitxer "MaintPunt3d.pse":

```
n1 <- crear Punt3d()
```

És cert, ja que *n1* s'ha declarat com a tipus enter, i la instrucció crear retorna una referència a una nova instància de la classe *Punt3d*.

El següent error es troba a la línia 17:

```
p1.sumar_simple(n1,23)
```

Si comprovem la declaració de la funció en el fitxer "Punt3d.pse", s'observa que és una funció privada que només apareix en el bloc implementació:

```
implementacio
                ...
      funcio sumar_simple(n1: enter,n2: enter) retorna r: enter
                r <- n1 + n2
                retorna r
      ffuncio
                ...
fimplementacio
```

Per tant, aquest error també ha estat detectat perfectament per el traductor. Finalment, l'últim error del fitxer apareix a la línia 18:

```
Punt3d.sumar(p1,p2)
```

Punt3d és el nom de la classe, i per tant amb aquesta notació només podem accedir a mètodes o atributs de classe. Però si comprovem la definició de *sumar*:

```
interficie
      ...
      metodes
                ...
                metode sumar(p1: Punt3d,p2: Punt3d) retorna pf: Punt3d
                ...
      fmetodes
      ...
finterficie
```

És un mètode públic de la instància, i per tant no pot ser accedit amb el nom de la classe, sino

que ha d'ésser accedit amb el nom d'alguna instància.

Per tant, el traductor ha reportat correctament els errors existents en el fitxer. Es pot observar com es comporta correctament amb les classes, adequant-se correctament al paradigma d'orientació a objectes.

6.3 Exemple 3: llibreria matemàtica

Per tal de demostrar la utilitat del traductor, s'ha demanat a un company que provés d'implementar algun programa en pseudocodi, per tal de provar el traductor. El resultat fou una classe que implementa una llibreria matemàtica amb les següents funcions: suma, resta, multiplicació, divisió, potència i factorial. A més, s'ha implementat dos fitxers més: l'algorisme principal i una classe d'interfície amb l'usuari que incorpora un menú per operar amb la classe matemàtica. A continuació es mostrarà el codi, per observar una aplicació real del llenguatge pseudocodi i del traductor.

Fitxer “BasicOperations.pse”

```
/** Classe llibreria que implementa les funcions matematiques
      Autor: Frederic Garcia
      Nota: Creat expressament per el CPC.
*/

classe BasicOperations
interficie
   metodes
      constructor BasicOperations()
      metode suma(ent1: enter, ent2: enter) retorna resultat: enter
      metode resta(ent1: enter, ent2: enter) retorna resultat: enter
      metode multiplica(ent1: enter, ent2: enter) retorna resultat: enter
      metode divideix(ent1: enter, ent2: enter) retorna resultat: enter
      metode potencia(base: enter, exponent: enter) retorna resultat: enter
      metode factorial(ent1: enter) retorna resultat: enter
   fmetodes
finterficie
implementacio
      var

      fvar
      constructor BasicOperations()
                // No fa res
```

```
fconstructor


metode suma(ent1: enter, ent2: enter) retorna resultat: enter
          resultat <- ent1 + ent2
          retorna resultat
fmetode

metode resta(ent1: enter, ent2: enter) retorna resultat: enter
          si (ent1 > ent2) llavors
                    resultat <- ent1 - ent2
          altrament
                    resultat <- -1
          fsi
          retorna resultat
fmetode

metode multiplica(ent1: enter, ent2: enter) retorna resultat: enter
          resultat <- ent1 * ent2
          retorna resultat
fmetode

metode divideix(ent1: enter, ent2: enter) retorna resultat: enter
          si (ent1 > ent2) llavors
                    resultat <- ent1 div ent2
          altrament
                    resultat <- -1
          fsi
          retorna resultat
fmetode

metode potencia(base: enter, exponent: enter) retorna resultat: enter
     var
          num: enter
     fvar

     si (exponent=0) llavors
          // Qualsevol nombre elevat a zero es 1
          resultat <- 1
     altrament
          // Inicialitzarem el resultat amb la base
          resultat <- base
          // Farem tantes multiplicacions com indiqui l'exponent
          per num<-2 fins exponent fer
               resultat <- resultat * base
          fper
     fsi
```

```
            retorna resultat
      fmetode

      metode factorial(ent1: enter) retorna resultat: enter
            var
                  aux: enter
            fvar

            resultat <- 1
            aux <- ent1
            // Comprovem que l'entrada sigui vàlida
            si (aux < 0) llavors
                  // No existeix el factorial de números negatius
                  resultat <- -1
            altrament
                  // Calculem el factorial del número entrat
                  si ((aux=0) o (aux=1)) llavors
                        // No cal fer res ja que resultat = 1
                  altrament
                        mentre (aux > 1) fer
                              resultat <- resultat * aux
                              aux <- aux-1
                        fmentre
                  fsi
            fsi

            retorna resultat
      fmetode
fimplementacio
fclasse
```

Fitxer “CalcInterficie.pse”

```
/** Classe principal: Interficie amb l'usuari
      Autor: Frederic Garcia
      Nota: Creat expressament per el CPC.
*/

classe CalcInterficie
interficie
      metodes
                  constructor CalcInterficie()
                  metode principal()
      fmetodes
finterficie
implementacio
      var
```

```
     math_lib: BasicOperations
fvar


funcio netejaPantalla()
     var
          e: enter
     fvar
     per e<-1 fins 28 fer escriureln("") fper
ffuncio
funcio pausa()
     var
          c: caracter
     fvar
     escriureln("Prem qualsevol tecla per continuar")
     c <- llegir()
ffuncio
funcio escriureMenu()
// Escriu el menu per pantalla
     netejaPantalla()
     escriureln("1. Suma de dos enters")
     escriureln("2. Resta de dos enters")
     escriureln("3. Multiplicacio de dos enters")
     escriureln("4. Divisio entera de dos enters")
     escriureln("5. Potencia")
     escriureln("6. Factorial d'un enter")
     escriureln("7. Sortir")
ffuncio
funcio suma()
// Realitza una suma de dos enters
     var
          ent1,ent2,result: enter
     fvar
     netejaPantalla()
     escriureln("SUMA de dos enters")
     escriureln("------------------")
     escriureln("")
     escriure("Entra el primer enter: ")
     ent1 <- llegir()
     escriure("Entra el segon enter: ")
     ent2 <- llegir()
     result <- math_lib.suma(ent1,ent2)
     escriureln("")
     escriure("El resultat es: ")
     escriure(result)
     escriureln("")
     pausa()
```

```
ffuncio
funcio resta()
// Realitza una resta entre dos enters
     var
          ent1,ent2,result: enter
     fvar
     netejaPantalla()
     escriureln("RESTA de dos enters")
     escriureln("-------------------")
     escriureln("")
     escriure("Entra el primer enter: ")
     ent1 <- llegir()
     escriure("Entra el segon enter: ")
     ent2 <- llegir()
     result <- math_lib.resta(ent1,ent2)
     escriureln("")
     si (result = -1) llavors
          escriureln("No s'han entrat els valors correctament.")
          escriureln("NOTA: El primer valor ha de ser mes gran o
igual que el segon.")
     altrament
          escriure("El resultat es: ")
          escriure(result)
          escriureln("")
     fsi
     pausa()
ffuncio
funcio multiplica()
// Realitza un producte entre dos enters
     var
          ent1,ent2,result: enter
     fvar
     netejaPantalla()
     escriureln("MULTIPLICACIO de dos enters")
     escriureln("---------------------------")
     escriureln("")
     escriure("Entra el primer enter: ")
     ent1 <- llegir()
     escriure("Entra el segon enter: ")
     ent2 <- llegir()
     result <- math_lib.multiplica(ent1,ent2)
     escriureln("")
     escriure("El resultat es: ")
     escriure(result)
     escriureln("")
     pausa()
ffuncio
```

```
funcio divideix()
// Realitza la divisio entera entre dos enters
     var
          ent1,ent2,result: enter
     fvar
     netejaPantalla()
     escriureln("DIVISIO entera de dos enters")
     escriureln("----------------------------")
     escriureln("")
     escriure("Entra el primer enter: ")
     ent1 <- llegir()
     escriure("Entra el segon enter: ")
     ent2 <- llegir()
     result <- math_lib.divideix(ent1,ent2)
     escriureln("")
     si (result = -1) llavors
          escriureln("No s'han entrat els valors correctament.")
          escriureln("NOTA: El primer valor ha de ser mes gran o
igual que el segon.")
     altrament
          escriure("El resultat es: ")
          escriure(result)
          escriureln("")
     fsi
     pausa()
ffuncio
funcio potencia()
// Calcula la potencia d'un enter elevat a un altre enter
     var
          base,exponent,result: enter
     fvar
     netejaPantalla()
     escriureln("POTENCIA")
     escriureln("--------")
     escriureln("")
     escriure("Entra la base: ")
     base <- llegir()
     escriure("Entra l'exponent: ")
     exponent <- llegir()
     result <- math_lib.potencia(base,exponent)
     escriureln("")
     escriure("El resultat es: ")
     escriure(result)
     escriureln("")
     pausa()
ffuncio
funcio factorial()
```

```
          // Calcula el factorial d'un enter
               var
                    ent1,result: enter
               fvar
               netejaPantalla()
               escriureln("FACTORIAL")
               escriureln("---------")
               escriureln("")
               escriure("Entra el nombre a calcular-ne el factorial: ")
               ent1 <- llegir()
               result <- math_lib.factorial(ent1)
               escriureln("")
               si (result = -1) llavors
                    escriureln("No s'ha entrat el valor correctament.")
                    escriureln("NOTA: No es pot calcular el factorial d'un
nombre negatiu.")
               altrament
                    escriure("El resultat es: ")
                    escriure(result)
                    escriureln("")
               fsi
               pausa()
          ffuncio
          constructor CalcInterficie()
               math_lib <- crear BasicOperations()
          fconstructor
          metode principal()
          // Metode principal: espera l'entrada de l'usuari i fa crides
          // a les funcions, depenent de l'entrada.
               var
                    ent: enter
                    continua: boolea
               fvar
               continua <- cert
               mentre continua fer
                    escriureMenu()
                    escriure("Introdueix l'opcio: ")
                    ent <- llegir()
                    cas
                         [] ent=1 -> suma()
                         [] ent=2 -> resta()
                         [] ent=3 -> multiplica()
                         [] ent=4 -> divideix()
                         [] ent=5 -> potencia()
                         [] ent=6 -> factorial()
                         [] ent=7 -> continua <- fals
                    fcas
```

```
            fmentre
        fmetode
    fimplementacio
    fclasse
```

Fitxer “calculator.pse”

```
    /** Algorisme que simula una calculadora amb operacions basiques i no tan basiques ;)
        Autor: Frederic Garcia
        Nota: Creat expressament per el CPC.
    */

    algorisme calculator
    var
        interf: CalcInterficie
    fvar
        interf <- crear CalcInterficie()

        interf.principal()
    falgorisme
```

Si executem el traductor per poder executar el programa, veiem que es tradueix tot sense donar cap error. Algunes pantalles de l'execució del programa:

```
1. Suma de dos enters
2. Resta de dos enters
3. Multiplicacio de dos enters
4. Divisio entera de dos enters
5. Potencia
6. Factorial d'un enter
7. Sortir
Introdueix l'opcio:
```

Imatge 6.3 Captura del menú del programa que implementa la llibreria matemàtica

```
FACTORIAL
---------

Entra el nombre a calcular-ne el factorial: 4

El resultat es: 24
Prem qualsevol tecla per continuar
```

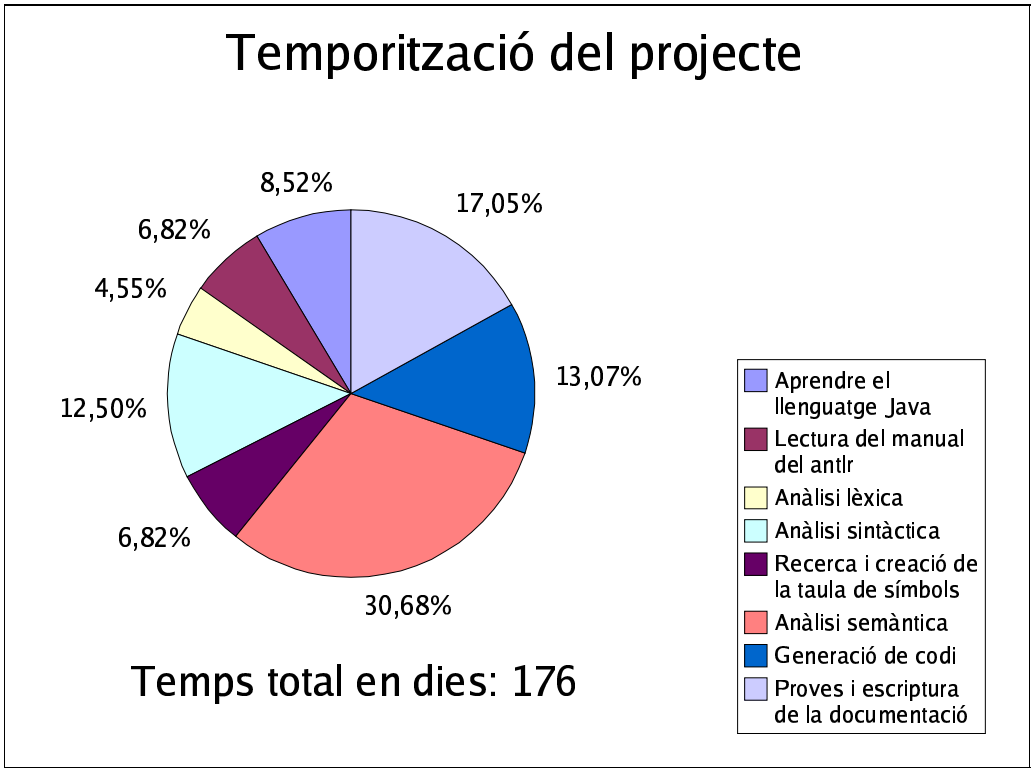
Imatge 6.4 Captura de la funció factorial del programa que implementa la llibreria matemàtica

7. Aspectes finals

7.1 Temporització orientativa

A continuació es mostrarà una temporització orientativa del que ha costat realitzar el projecte. Cal dir que el projecte s'ha realitzat en les estones lliures que tenia mentre realitzava paral·lelament tres assignatures. Per tant, s'ha de tenir en compte que aquesta temporització és aproximada, i només serveix per tenir una idea del cost de la seva implementació.

Tot seguit, es mostrarà un gràfic en el que es detalla el temps dedicat a cada tasca:



Imatge 7.1 Temporització orientativa del projecte

Com es pot observar, l'etapa més llarga ha sigut la realització de l'anàlisi semàntica, seguida de la fase de proves i escriptura de la documentació. La recerca i creació de la taula de símbols també ha sigut una tasca a destacar, ja que la manipulació dels àmbits era un aspecte clau per tal que funcionés la posterior anàlisis semàntica.

7.2 Problemes

Com qualsevol projecte, aquest no ha estat absent de problemes, i a continuació relatarem els més importants que han sorgit.

Els indeterminismes provocats per el salt de línia. Això s'ha solucionat no ignorant el comentari simple d'una línia (que conté el retorn de carro), i definint un predicat separador d'instrucció com el salt de línia i el comentari. D'aquesta manera es permetia posar un comentari després d'una instrucció, i la definició del predicat ajudava a treure els indeterminismes en els altres predicats.

Els llenguatges orientats a objectes aporten grans avanços en la tecnologia de la programació: faciliten la realització i manteniment d'aplicacions grans i ajuden a la reutilització de codi. Però compliquen molt la tasca de construir compiladors/traductors per aquests, ja que s'ha de realitzar moltes comprovacions que no es feien pas en la programació estructurada. Sobretot el tema de l'herència ha sigut una mica complicat de tractar. Si un mètode/atribut no es trobava declarat en l'àmbit actual, s'ha de comprovar si és una subclasse, i en cas afirmatiu comprovar si està declarat en la superclasse corresponent. I això esdevé un bucle fins a trobar la declaració o una classe que no tingui cap superclasse.

L'ús de classes externes ha sigut complicat de tractar. Quan es declara una variable com a referència d'una classe, si la definició o interfície d'aquesta no està en la taula de símbols, cal intentar carregar-la del fitxer. Per carregar la seva interfície, el que es fa és declarar un nou objecte de tipus CpcParser (el *parser* del traductor) i s'intenta traduir el fitxer. En acabat aquesta traducció, s'accedeix a la taula de símbols generada i s'emmagatzema en la taula de símbols del procés traductor original i es continua la compilació. Aquest model de treball és una simplificació de l'actuació real del **Java**. Té l'avantatge que sempre es comprova que sigui correcte la classe externa, i sempre tenim la seva definició més actual, ja que si es realitzen canvis posteriors, sempre s'actualitzaran al haver-se de traduir obligatòriament. Malauradament presenta inconvenients de rendiment. En codis fonts complexes on intervingui l'herència, la mateixa classe externa pot ésser traduïda moltes vegades, mentre que el **Java** només l'hagués traduït un cop. No obstant, la pèrdua de rendiment no és massa notòria, i per tant s'ha continuat amb el model implementat.

7.3 Treballs futurs

En aquest apartat s'exposen possibles millores al projecte presentat, però cal remarcar que la realització d'aquestes millores faria que el traductor esdevingués a un de gran nivell. D'altra banda, el treball complet seria impossible imposar-lo per la seva gran complexitat i per la descompensació entre el treball i el temps que suposa fer-ho i la poca quantitat de crèdits de valor d'un treball final de carrera. Les millores es relaten a continuació.

- Actualment el traductor cada vegada que es declara una classe que no troba en la taula de símbols, la *parseja* per tal de trobar la interfície amb els membres públics. Però el procés de *parseig* pot decaure el rendiment en programes molt grossos, ja que cada vegada s'escriu el fitxer **Java** resultant en el disc. Es podria millorar comprovant si ja està *parsejat* i compartint la definició de la classe entre els diferents fitxers que es tracten en el moment de la compilació.
- Les taules només incorporen un mètode per saber la llargària d'aquestes. Però en la definició formal del pseudocodi també es permet saber les dimensions de la taula i la llargària de cada dimensió per separat i seria interessant la seva implementació. La traducció d'aquestes taules no es podria fer directament en **Java**, sinó que es tindria que fer en una classe a part, ja que el **Java** no permet fer-ho amb les taules proporcionades per el llenguatge.
- Millorar l'entrada i sortida per poder manipular fitxers, ratolí, impressora i altres perifèrics.
- Crear una llibreria de classes útils que incorporés classes com: cadena, seqüència (utilitzada en les pràctiques de programació de primer d'ETIS i d'ETIG),...
- Implementar l'assignació múltiple acceptada en el pseudocodi. No es pot fer directament per què el **Java** no l'accepta.
- Es podria crear diferents *back-end* per tal de traduir el codi final a diferents llenguatges. Només caldria canviar la classe **CpcGenJava** per una classe que generés per exemple codi en **C++** i canviar la classe d'entrada i sortida per defecte. D'aquesta manera el traductor seria molt més versàtil ja que es podrien generar molt més llenguatges partint del pseudocodi.

7.4 Conclusió

Les conclusions que he tret d'aquest projecte són les que s'expliquen a continuació.

El fet de disposar d'eines que realitzen la part de feina més mecànica de la generació de compiladors facilita el treball del programador, que no ha de perdre el temps implementant codi sistemàticament, sinó que s'ha de preocupar de les parts que requereixen més creativitat. Aquestes eines permeten, per tant, programar en un nivell més elevat. No obstant, sempre és interessant la característica de poder-hi inserir codi en algun llenguatge de programació, ja que fa que la eina sigui molt més potent.

L'anàlisi lèxica és la de més fàcil implementació, mentre que l'anàlisi sintàctica és una mica més complicada per què és la que defineix estrictament el llenguatge i s'ha d'anar en compte en no caure en indeterminismes quan es defineix la gramàtica. L'anàlisi semàntica és l'etapa més difícil, ja que s'ha de realitzar comprovacions de tipus, de visibilitat, herència, entre d'altres. La generació de codi **Java** només té el problema de que el text *parsejat* ha d'ésser passat entre fase i fase del traductor per tal de no perdre valors que hem d'escriure en aquesta fase, com expressions aritmètiques, identificadors, etc.

Les gramàtiques en la que el separador d'instruccions és un salt de línia, com el **Visual Basic** o el propi pseudocodi, enlloc de l'estès punt i coma complica molt més la definició del llenguatge.

Que la programació orientada a objectes requereix d'uns compiladors molt més potents que la programació estructurada per a poder tractar adequadament termes complexos com la sobrecàrrega de mètodes o la herència. Sobretot, propietats com l'herència compliquen molt la fase semàntica del compilador, ja que s'han de realitzar moltes més comprovacions que no existeixen pas ens els llenguatges de programació estructurada tradicionals.

Finalment, com a valoració més personal, cal dir que he aprés amb exactitud quins són els diferents objectius de cada fase en què s'estructura un compilador/traductor. He aprofundit molt més en la construcció de cada etapa d'un compilador que no pas ho vaig fer en les pràctiques de l'assignatura optativa de *compiladors*.

7.5 Bibliografia

APPEL w. Andrew, *Modern Compiler implementation in Java*, Cambridge University Press, 1998.

BENNET, J.P. *Introduction to compiling techniques*, Londres: McGraw-Hill Book Company Europe, 1990.

PARR Terence, *ANTLR Website* a http://www.antlr.org

The LEX & YACC Page a http://dinosaur.compilertools.net/

KAPLAN Ian, *The ANTLR Parser Generator* a http://www.bearcave.com/software/antlr/index.html

KAPLAN Ian, *Architecture of a Java Compiler a* http://www.bearcave.com/software/java/comp_arch.html

MASSOT BAYÉS Marc, *Documents de Java* a http://ima.udg.es/~marc/docs_java/

7.6 Altres recursos d'interès

- Projecte mono: Implementació *open source* de la plataforma de desenvolupament .NET de *Microsoft*.

  Enllaç: http://go-mono.com/index.html

En el capítol 4, s'esmenta que la plataforma *mono* utilitza un analitzador lèxic implementat a mà, enlloc d'utilitzar el que es genera amb alguna eina de creació de compiladors. Concretament és el compilador per el llenguatge **C#** i la informació es troba en:

Enllaç: http://go-mono.com/c-sharp.html

- Projecte GNU GCC: Implementació de compiladors *open source* per diferents llenguatges i diferents plataformes de maquinari.

  Enllaç: http://www.gnu.org/software/gcc/gcc.html

Es poden baixar els codi font de qualsevol dels compiladors, i es pot estudiar al seu codi, així com col·laborar en el seu desenvolupament.

- Història del ANTLR: primer anomenat PCCTS per a la tesis de Terence Parr (incloïa les eines ANTLR i DLG) i després s'ha passat a anomenar ANTLR.

  Enllaç: http://www.antlr.org/history.html

7.7 Agraïments

M'agradaria dedicar aquesta pàgina a un agraïment a totes aquelles persones que m'han ajudat, recolzat, motivat en la realització d'aquest projecte:

Josep Suy El meu tutor de projecte i qui m'ha ajudat en tots els problemes que he tingut.

Frederic Garcia Que ha provat el traductor i ha escrit un programa per a ell.

Susanna Encesa Sempre m'ha recolzat moralment i he tingut el seu suport incondicional.