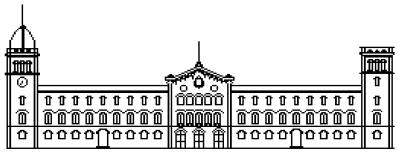


Treball fi de carrera

ENGINYERIA TÈCNICA EN INFORMÀTICA DE SISTEMES

Facultat de Matemàtiques
Universitat de Barcelona

TRADUCCIÓ D'UN PSEUDOCODI

Susana Crespo Ahedo

Director: Jaume Timoneda Salat
Realitzat a: Departament de Matemàtica Aplicada i Anàlisi. UB

Barcelona, 4 de juliol de 2007

ÍNDEX

1. Introducció

2. Objectius

3. Traductors i compiladors

3.1. Traductor
3.2. Compilador
    3.2.1. Fase Anàlisi
    3.2.2. Fase Síntesi
    3.2.3. Fase Optimització
    3.2.4. Exemples
    3.2.5. Taula de símbols
    3.2.6. Estructura
3.3. Traductor i compilador del projecte

4. Generador LEX

4.1. Esquema d'us
4.2. Esquema general
    4.2.1. Secció de definicions
    4.2.2. Secció de regles
    4.2.3. Secció de rutines
    4.2.4. Criteri de selecció
    4.2.5. Enmagatzament sentencia acoplada
    4.2.6. Patró per defecte
4.3. Obtenció i execució de l'analitzador
4.4. Expressions regulars
    4.4.1. Operació opcionalitat
    4.4.2. Operació repetició una o més vegades
    4.4.3. Operació repetició limitada
    4.4.4. Corxets
    4.4.5. Metacaracters
    4.4.6. Seqüències d'escape

4.4.7. El caràcter universal
4.4.8. Subexpressions amb nom
4.4.9. Relació de metacaracters

4.5. Definició i traducció de les paraules reservades del `pseudocodi`
3.5.1. Paraules amb traducció directe
3.5.2. Paraules amb traducció directa i presencia important
3.5.3. Paraules amb traducció complexa

5. L'script compilador

5.1. Definició de l'script creat
5.2. Comandes internes utilitzades
5.2.1. `getopts`
5.2.2. `sed`
5.2.3. `indent`
5.2.4. `gcc`

6. Resultats

7. Conclusions

7.1. Fase d'estudi
7.2. Disseny i implementació
7.3. Treball futur

8. Bibliografia

9. Annexos

9.1. Codi `Lex`
9.2. Codi script

AGRAIMENTS

No puc esmentar a tots, però si que m'agradaria reconèixer específicament el valor d'algun d'ells:

A la meva família, per el seu suport i ànims que m'han acompanyat en tota la meva carrera d'estudiant.

A la meva parella Iván, per creure sempre en mi, pels seus ànims, per la seva ajuda i sobretot per la seva infinita paciència.

A Desirée, més que una companya i més que una amiga, qui m'ha ajudat en tots els anys de carrera i qui m'ha ensenyat moltes coses.

Als meus companys d'universitat: Sara, Sandra, Imma, Miguel, Marc, Capelo, Dani, Julian, Yoli, Victor, Oscar, Mary, Ning i tota la resta per fer d'aquest anys universitaris els millors de la meva vida.

Y per últim, i molt agraïda, al meu tutor de projecte Jaume Timoneda, qui sempre m'ha estat ajudant, donant informació, ensenyant i revisant molt periòdicament el meu treball.

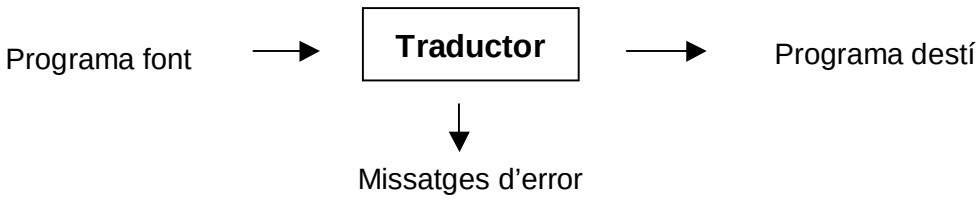
1. INTRODUCCIÓ

Un dels principals mecanismes de comunicació entre un ordinador i una persona ve donada per la tramesa i recepció de missatges de tipus textual: l'usuari escriu una ordre mitjançant el teclat, i l'ordinador l'executa tornant com a resultat un missatge informatiu sobre les accions portades a terme.

Encara que l'evolució dels ordinadors es troba dirigida actualment cap a l'ús d'interfícies d'usuari noves i ergonòmiques (com el ratolí, les pantalles tàctils, etc.), podem dir que gairebé totes les accions que l'usuari realitza sobre aquestes interfícies es tradueixen abans o després a seqüències de comandes que són executades com si haguessin estat introduïdes per teclat. Per un altre costat, i des del punt de vista d'un informàtic, moltes de les accions que es veurà obligat a desenvolupar en el transcurs de la seva carrera professional, hauran de veure amb traductors: la programació, la creació de fitxers `batch`, la utilització d'un intèrpret de comandes, etc.

Un traductor és un programa que tradueix o converteix des d'un text o programa escrit en un llenguatge font fins un text o programa escrit en un llenguatge destí. És important destacar la velocitat en la qual avui en dia es fan. En la dècada de 1950, es va considerar els traductors com a programes notablement difícils d'escriure. El primer compilador de FORTRAN, per exemple, va necessitar per a la seva implementació 18 anys de treball en grup. Fins que va aparèixer la teoria d'autòmats no es va poder accelerar ni formalitzar la creació de traductors.

L'esquema d'un traductor és:



Hi ha una gran quantitat de situacions en les que pot ser molt útil conèixer com funcionen les diferents parts d'un compilador, especialment aquella que s'encarrega de trossejar els text fonts i convertir-los en frases sintàcticament valides. Aplicacions de la construcció de traductors poden ser la creació de preprocessadors per llenguatges que no tenen (per exemple, per treballar fàcilment amb SQL en C, es pot fer un preprocessador per introduir SLQ immers), o inclòs la conversió de caràcter ASCII 10 (LF) en "
" de HTML per passar text a la web.

Des dels seus orígens, hi ha hagut un "buit" entre la forma d'expressar-se de les persones i de les màquines. Els traductors han intentat escurçar aquest buit per facilitar-li les coses a les persones, el que ha portat a aplicar la teoria d'autòmats a diferents camps i a àrees concretes de la informàtica.

Tenim diferents tipus de traductors:

- Traductors de l'idioma.
- Compiladors.
- Intèrprets.
- Preprocessadors.
- Intèrprets de comandes.
- Ensambladors i Macroensambladors.
- Conversors font – font.
- Compilador creuat.

2. OBJECTIUS

L'objectiu és realitzar un compilador que transformi un algoritme escrit en un llenguatge específic per un algoritme en codi C. El llenguatge del fitxer font, que tindrà extensió .psc es un pseudocodi (per tenir més informació del pseudocodi consultar la bibliografia).

Dispondrá del següents components:

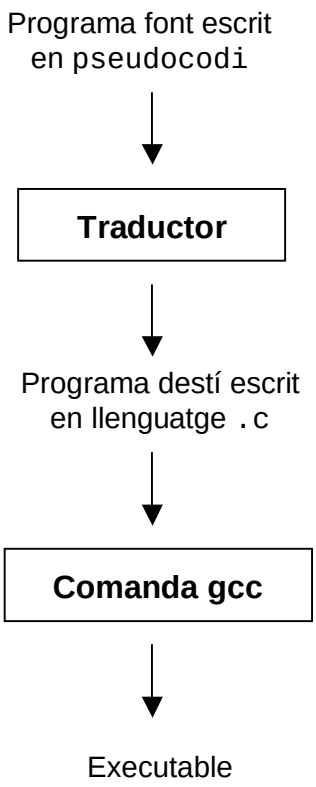
- Analitzador lexicogràfic.
- Generació de codi.
- Compilació de l'arxiu .c.

El pseudocodi esta orientat, no solament a promoure la lògica elemental d'una programació estructurada, sinó també a proporcionar un model de llenguatge simple per a la descripció no ambigua d'algorismes de baix nivell, per la seva fàcil comprensió és més ràpid d'aprendre que a programar en C. Amb el traductor realitzat en aquest projecte permetrem a les persones que programin en aquest pseudocodi l'oportunitat de compilar els seus programes en un executable per poder-los provar i utilitzar.

En aquest treball s'utilitzarà l'eina Lex per fer l'anàlisi lèxic d'un text d'entrada (que serà un arxiu en pseudocodi) i poder convertir-lo en un text de sortida (que serà un arxiu .c). L'arxiu .c que obtindrem el compilarem amb l'ajuda de la comanda gcc, d'aquesta manera s'obtindrà un arxiu compilat en C amb la traducció i es podrà provar si el seu funcionament és correcte.

Aquesta traducció no contempla la part sintàctica del compiladors, és a dir, si el programa escrit en pseudocodi és incorrecte la seva traducció a C també ho serà. S'aconsegueix el compilador utilitzant la comanda gcc per compilar l'arxiu .c i si aquest no és correcte, si te algun error sintàctic, la pròpia comanda gcc serà la que s'encarregarà de donar els errors obtinguts de l'arxiu .c i serà l'usuari qui tindrà que interpretar aquests errors en C en el seu codi escrit en pseudocodi ja que les línies d'error i les descripcions dels errors es correspondran a l'arxiu C.

L’estructura final del projecte serà:



3. TRADUCTORS I COMPILADORS

3.1. Traductors

Com s'ha descrit a l'apartat d'introducció existeixen diferents tipus de traductors, per veure les diferencies entre ells es farà una petita menció del que cadascun d'ells fa:

- El *traductor d'idioma* tradueix d'un idioma donat a un altre, per exemple, un traductor d'anglès a Espanyol.

- El *compilador* és aquell traductor que té com a entrada una sentència en llenguatge formal i com sortida té un fitxer executable, és a dir, fa una traducció d'alt nivell a codi màquina.

- Un *intèrpret* és com un compilador, sol que la sortida és una execució. El programa d'entrada s'interpreta i executa alhora.

- Un *preprocessador* permet modificar el programa font abans de la compilació. Fa ús de macroinstruccions i directives.

- *L'Intèrpret de comandes* el que fa és traduir sentències simples a crides a programes d'una biblioteca. Són especialment utilitzats per Sistemes Operatius.

- *L' Ensamblador i el Macroensamblador* són els pioners dels compiladors, ja que en els principis de la informàtica, els programes s'escrivien directament en codi màquina, i els ensambladors estableixen una relació biunívoca entre cada instrucció i una paraula mnemotècnica, de manera que l'usuari escriu els programes fent ús dels mnemotècnics, i l'ensamblador s'encarrega de traduir-ho al codi màquina pur.

- El *Conversor font - font* passa un llenguatge d'alt nivell a un altre llenguatge d'alt nivell, per a aconseguir major portabilitat.

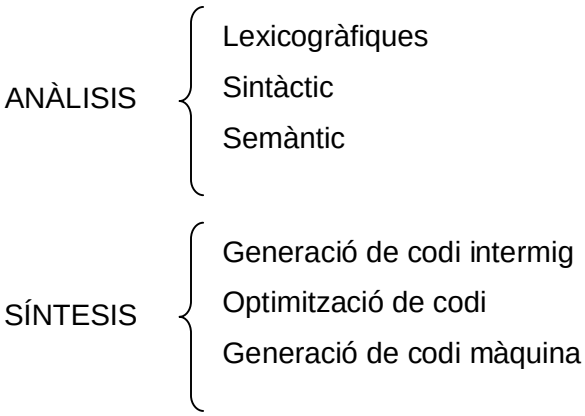
- I per últim un *Compilador creuat* és un compilador que obté codi per a executar en una altra màquina. S'utilitzen en la fase de desenvolupament de nous ordinadors.

3.2. Compiladors

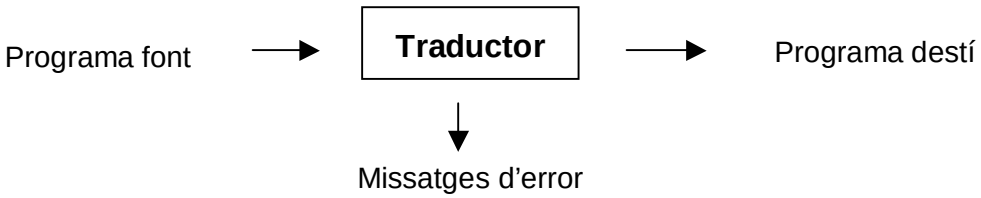
Una vegada hem vist els tipus de traductors que hi ha aprofundirem una mica més en l'estructura que té un compilador per entendre millor el seu funcionament:

Un compilador es divideix en dues fases:

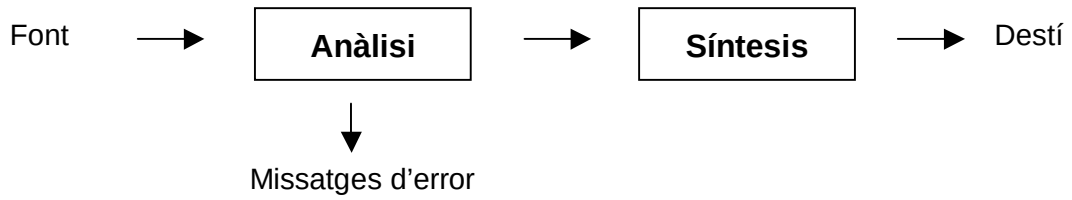
- Una part que analitza l'entrada i genera estructures intermèdies.
- Una altra part que sintetitza la sortida. Sobre la base de tals estructures intermèdies.



Fins a ara l'esquema d'un traductor era:



Depurem aquest esquema:



3.2.1. Fase Anàlisi

Bàsicament els objectius de la fase d'anàlisi són:

- Controlar la correcció del programa font.
- Generar estructures necessàries per a començar la síntesi.

Per a dur això, l'anàlisi consta de les següents tasques:

- Anàlisi Lexicogràfic : Divideix el programa font en els components bàsics: nombres, identificadors d'usuari (variables, constants, tipus, noms de procediments,...), paraules reservades, signes de puntuació., etc. A cada component li associa la categoria a la qual pertany.

- Anàlisi Sintàctic: Comprova que l'estructura dels components bàsics sigui correcta segons certes regles gramaticals.

- Anàlisi Semàntic: Comprova la resta, és a dir, tot el relacionat amb el significat, mirar els tipus, rangs de valors, existència de variables, etc.

En qualsevol dels tres anàlisis pot haver-hi errors.

3.2.2. Fase Síntesi

L'objectiu de la fase de síntesi consisteix en construir el programa objecte desitjat a partir de les estructures generades per la fase d'anàlisi.

Per a això realitza tres tasques fonamentals:

- Generació del codi intermedi: Genera un codi independent de la màquina. És fàcil fer pseudocompiladors i a més facilita l'optimització del codi.

- Generació del codi màquina: Crea un fitxer '`.*exe`' directament o un fitxer '`.obj`'. Aquí també es pot fer optimització pròpia del microprocessador.

- Fase d'optimització: L'optimització pot realitzar-se durant les fases de generació de codi intermedi i/o generació de codi màquina i pot ser una fase aïllada d'aquestes, o estar integrada amb elles.

3.2.3. Fase Optimització

L'optimització del codi intermedi ha de ser independent de la màquina. Amb freqüència, les fases s'agrupen en una etapa inicial *(Front-End)* i una etapa final *(Back- End)*.

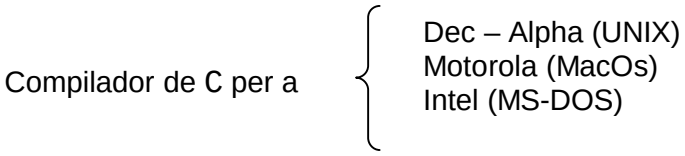
- L'etapa inicial comprèn aquelles fases, o parts de fases que depenen principalment del llenguatge font i que són en gran part independents de la màquina objecte. Aquí normalment s'introdueixen els anàlisis lèxics i sintàctics, la creació de la taula de símbols, l'anàlisi semàntic i la generació de codi intermedi.

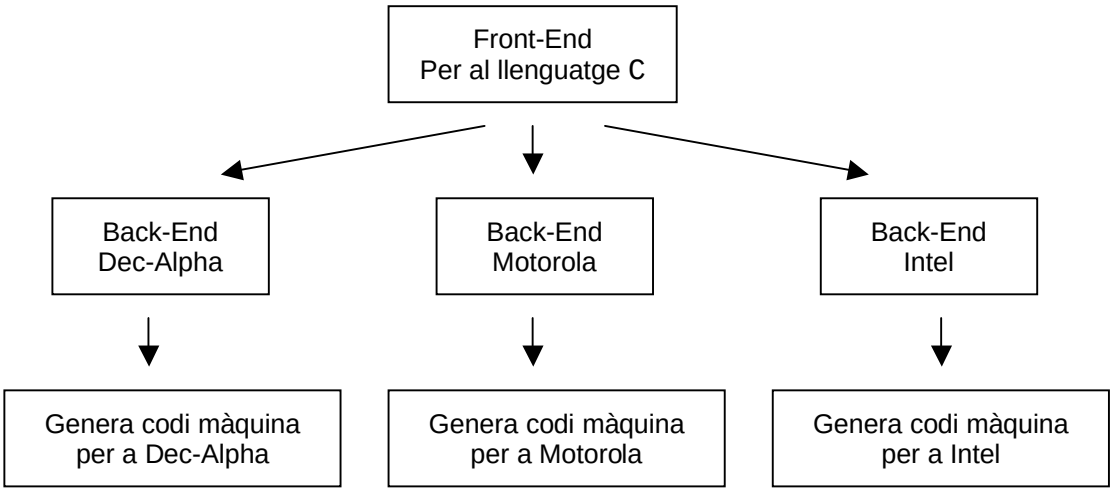
- L'etapa final inclou aquelles parts del compilador que depenen de la màquina objecte i, en general, aquestes parts no depenen del llenguatge font, sinó només del llenguatge intermedi. En l'etapa final, es troben aspectes de la fase d'optimització de codi, a més de la generació de codi, juntament amb el maneig d'errors necessari i les operacions amb la taula de símbols.

S'ha convertit en rutina el prendre l'etapa inicial d'un compilador i refer la seva etapa final associada per a produir un compilador per al mateix llenguatge font en una màquina distinta. També es pot compilar diversos llenguatges diferents en el mateix llenguatge intermedi i usar una etapa final comuna per a les distintes etapes inicials, obtenint així diversos compiladors per a una màquina.

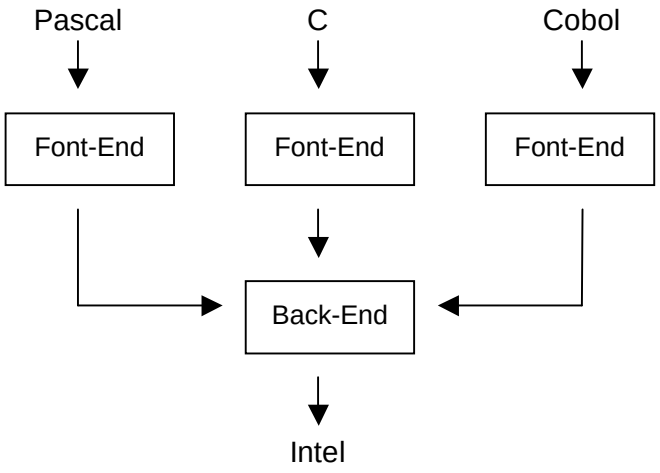
3.2.4. Exemples

1.- Suposem que volem crear un compilador del llenguatge C per a tres màquines diferents.

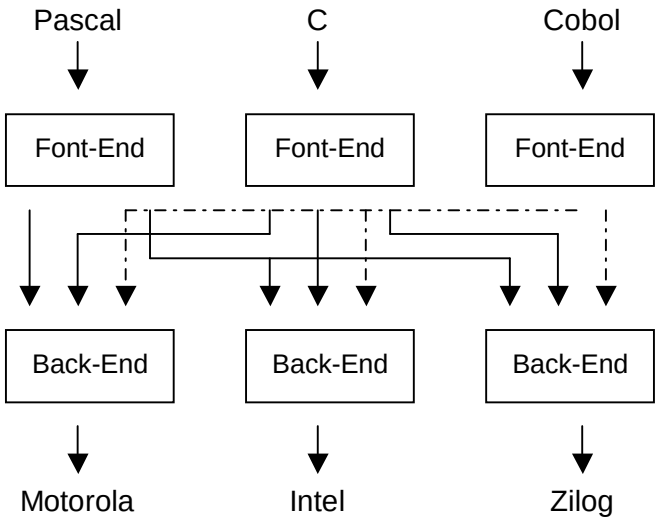




2.- També podem crear tres compiladors per a la mateixa màquina.



3.- Si volem tres compiladors per a tres màquines tindrem 9 compiladors en total.

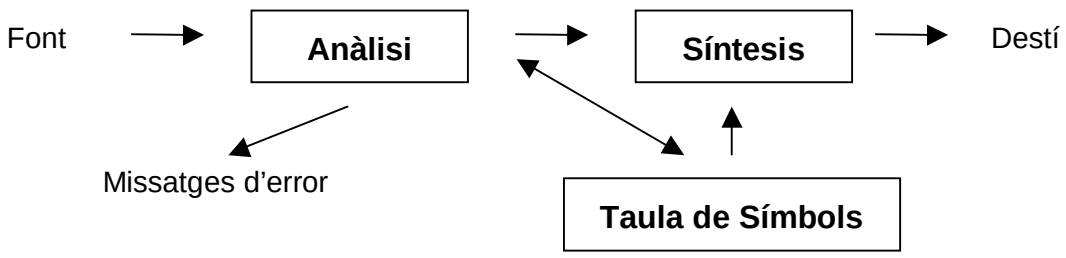


Una funció essencial d'un compilador és registrar els identificadors utilitzats en el programa font i reunir informació sobre els diferents atributs de cada identificador. Aquests atributs poden proporcionar informació sobre la memòria assignada a un identificador, el seu tipus, el seu àmbit,...

3.2.5. Taula de símbols

Posseeix informació sobre els identificadors definits per l'usuari, ja siguin constants, variables o tipus. Pot contenir informació de diversa índole, ha de fer-se de manera que no sigui uniforme. Fa funcions de diccionari de dades i la seva estructura pot ser una taula bash, un arbre binari de recerca, etc.

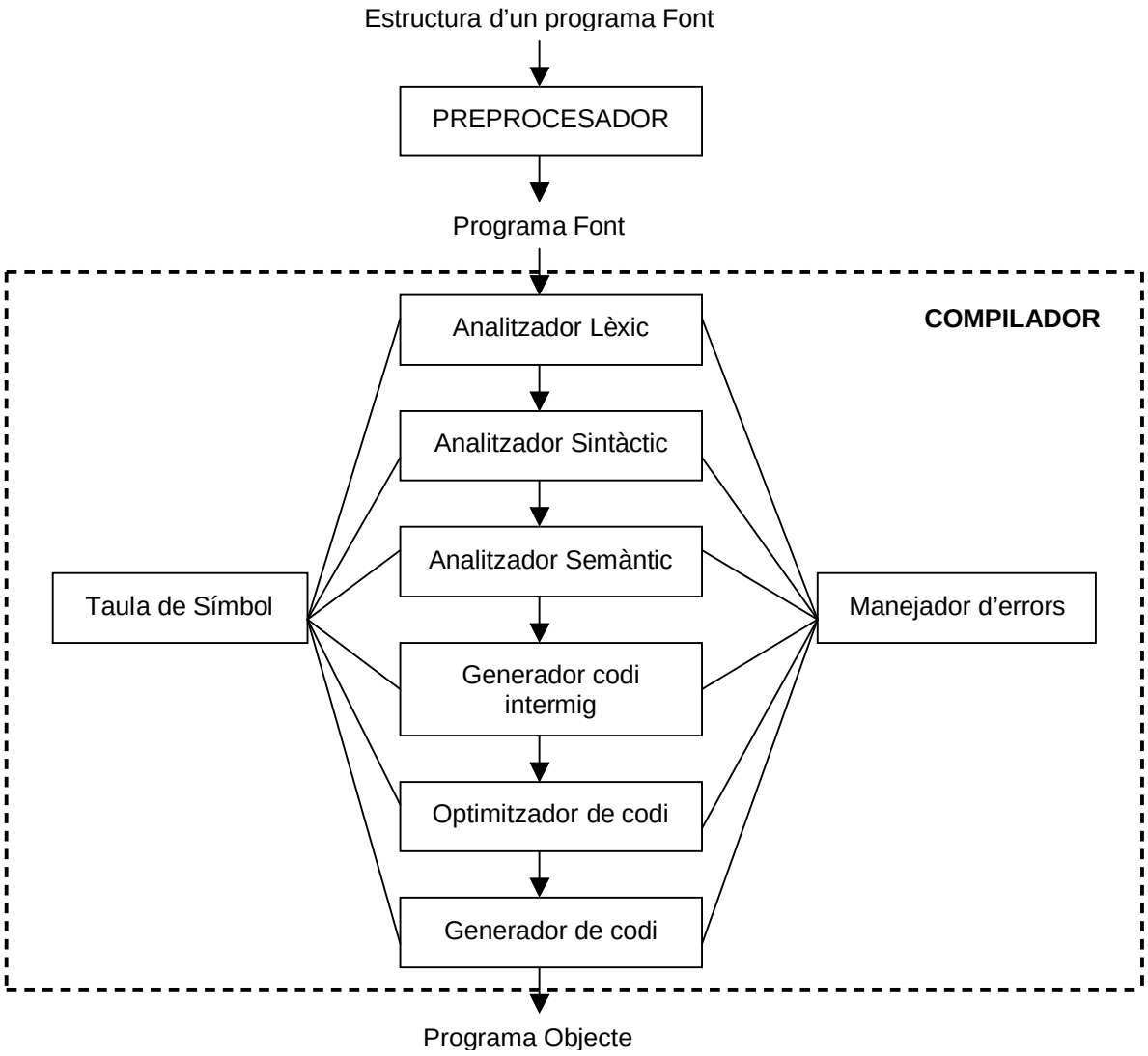
Amb tota aquesta informació, l'esquema definitiu d'un traductor es:



3.2.6. Estructura

Com s'ha explicat un compilador es divideix en dues fases, i cadascuna d'aquestes fases té una sèrie de tasques fonamentals, les quals transforma al programa font d'una representació en una altra.

L'estructura final es:



4.- EL GENERADOR LEX

Per fer la traducció del `pseudocodi` s'ha utilitzat l'eina `Lex` que ens permet analitzar les paraules contingudes en l'arxiu del `pseudocodi` i escriure les noves instruccions en l'arxiu de sortida. `Lex` és una de les eines més conegudes que s'utilitzen en la generació automàtica de compiladors dels llenguatges de programació. És una utilitat del sistema operatiu Unix i els programes que es produeixen estan escrits en llenguatge `C`.

A continuació es farà una descripció dels aspectes bàsics d'aquesta eina; en concret, es tracta de la seva aplicació per a l'obtenció de l'analitzador lexicogràfic dels traductors de llenguatges.

Un generador d'analitzadors és un programa que accepta com a entrada l'especificació de les característiques d'un llenguatge L i produeix com a sortida un analitzador per a L. L'especificació d'entrada pot referir-se a la lexicografia, la sintaxi o la semàntica; l'analitzador resultant serà útil per analitzar les característiques especificades.

E Especificació de les característiques del llenguatge L
A Analitzador para L

El generador `Lex` serveix, per generar analitzadors lexicogràfics per al seu aprofitament com a parts dels compiladors dels llenguatges de programació; aquests usos de `Lex` no són els únics, encara que sí són els que aquí es consideren principalment perquè són els que s'han utilitzat per aquest projecte.

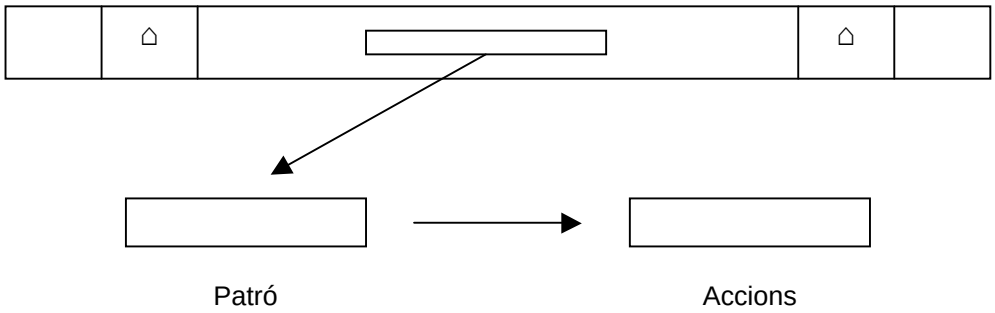
Quan s'empra el terme `Lex`, s'esmenten dos possibles significats:

a) Una notació per especificar les característiques lexicogràfiques d'un llenguatge de programació.

b) Un traductor d'especificacions lexicogràfiques.

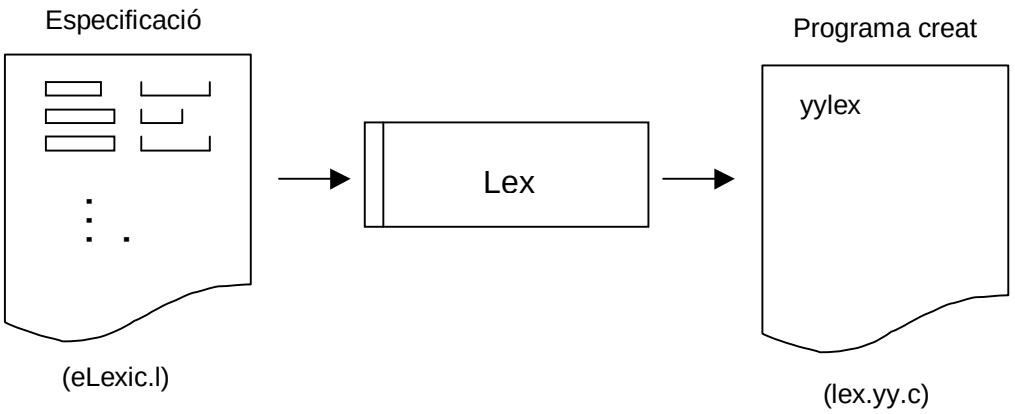
El problema que és pretén resoldre amb l'ajuda d'aquesta eina consisteix a trobar en un text gravat en un fitxer seqüències de caràcters que s'ajustin a unes determinades formes (a uns determinats models o patrons), i una vegada trobada una seqüència que s'ajusta a un patró procedir a la realització d'unes operacions que es tinguin associades a aquest patró o a la seva substitució.

Si ⌂ indica inici i final d'una entrada d'un arxiu escrit en pseudocodi:



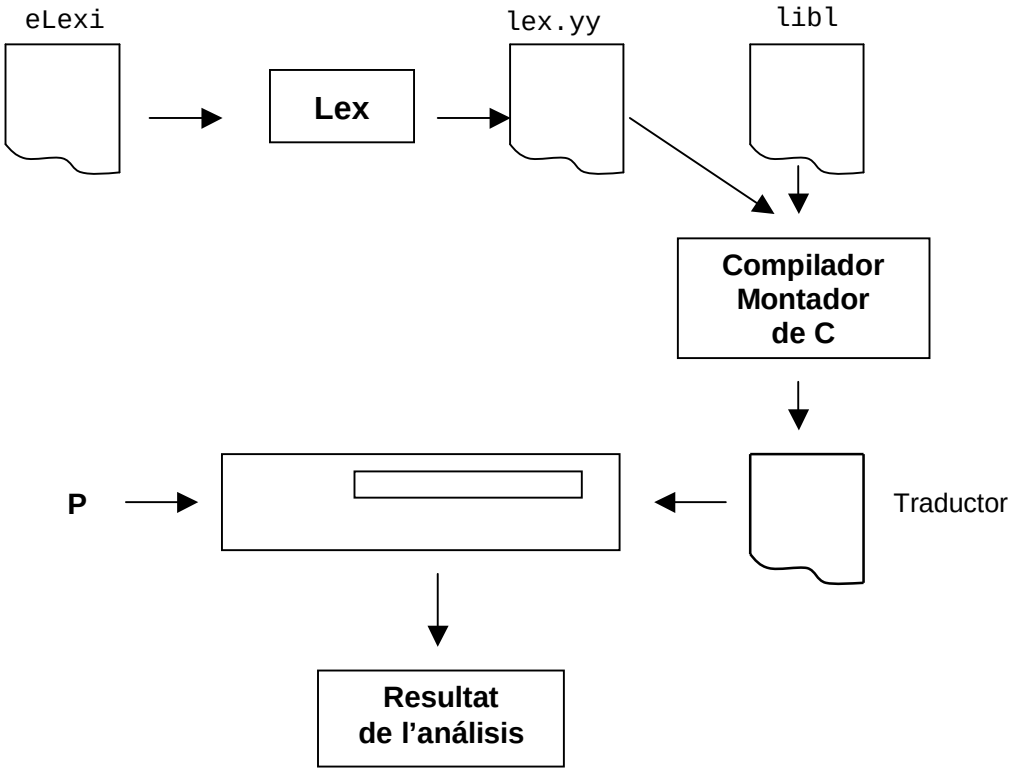
En la pràctica habitual solen tenir-se diversos patrons, cadascun d'ells amb els seus corresponents operacions associades.

El següent esquema mostra en què consisteixen l'entrada i la sortida per al programa traductor Lex. L'entrada a Lex és una especificació en la qual s'associen els patrons i les accions.



4.1. Esquema d'ús

L'esquema següent il·lustra la manera d'usar el generador Lex per obtenir un analitzador lèxic d'un llenguatge de programació L, i d'executar l'analitzador obtingut.



Els noms que apareixen en l'esquema signifiquen:

- **eLexic.l:** És l'especificació de les característiques lexicogràfiques del llenguatge L, escrita a Lex.

- **lex.yy.c:** És l'analitzador lexicogràfic de L generat per Lex; està constituït, en la seva part principal, per una funció escrita en C que realitza les tasques d'anàlisi lexicogràfica basant-se en autòmats regulars reconeixedors de la forma de les peces sintàctiques de L.

- **Libl:** És una llibreria associada a Lex que conté estructures de dades i funcions a les què és pot fer referència des del codi generat.

- **Traductor:** És l'analitzador generat; analitza les característiques lexicogràfiques especificades del llenguatge L; accepta com a entrada un programa escrit en L i comprova si està codificat segons les especificacions donades.

No és precís que els noms dels fitxers d'entrada per a Lex tinguin una extensió determinada; els noms dels fitxers generats per Lex són sempre els indicats, amb independència de quin sigui el nom dels fitxers d'entrada.

Les accions es descriuen mitjançant codi escrit en el llenguatge C a causa de que:

- El traductor Lex produeix un analitzador que és un programa escrit en C.
- El codi de les accions escrit en l'especificació d'entrada es trasllada de manera literal a la sortida, això és, queda incorporat al programa escrit en C.

La sortida produïda per Lex és un programa, la part principal del qual la constitueix una funció, de nom yylex, que realitzà l'anàlisi d'un text segons els patrons indicats en l'especificació de l'entrada. L'algoritme definit per la funció està representat en el següent cicle:

Mentre quedi text per analitzar:

- Acoblar un patró a partir del punt actual de l'entrada.
- Avançar en l'entrada fins a sobrepassar la seqüència acoblada.
- Executar l'acció associada al patró acoblat.

Lex genera una funció de nom yylex; perquè s'executi yylex ha de produir-se una cridada, des de distints llocs:

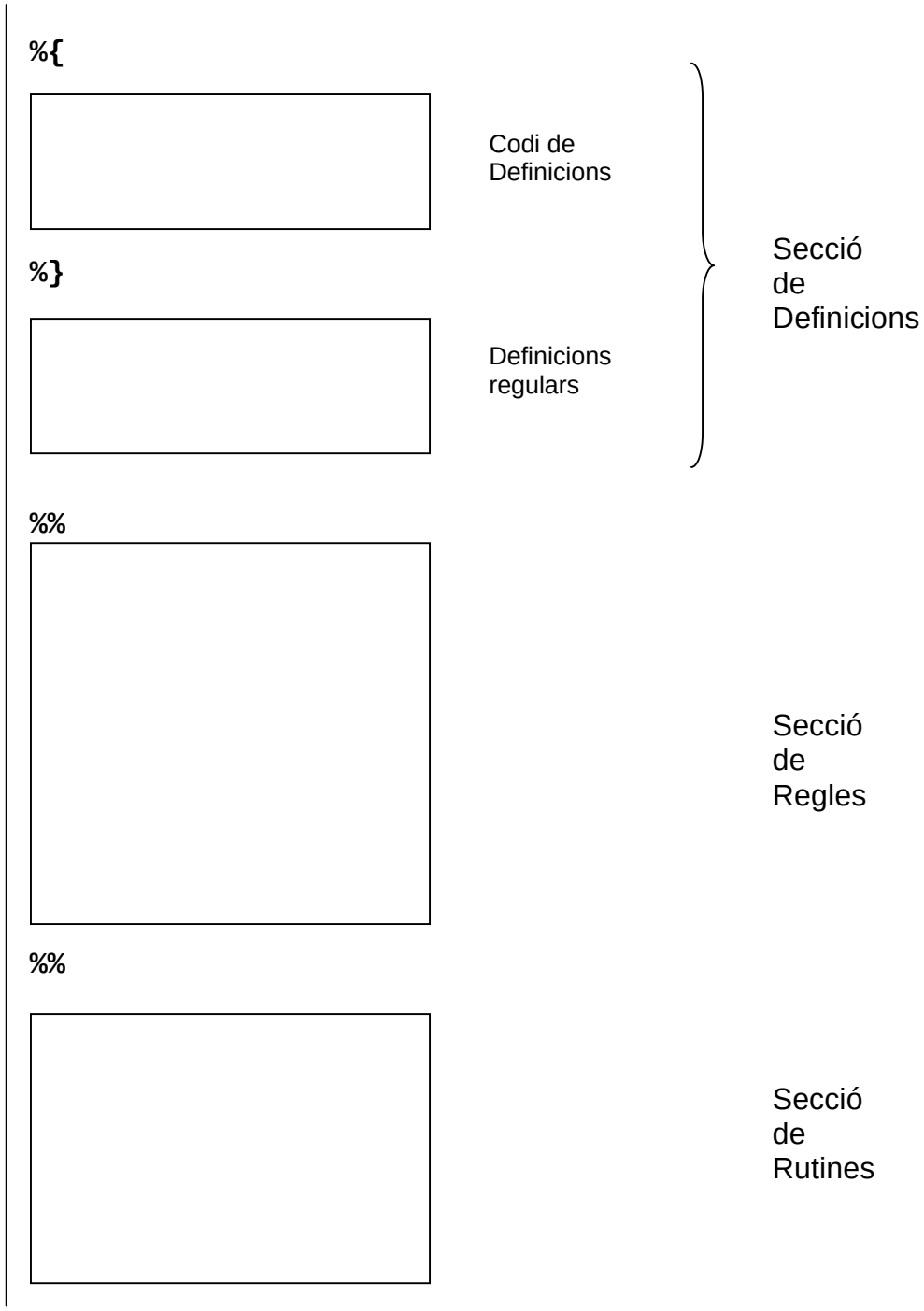
- Des de la funció que faci les vegades d'analitzador sintàctic, en el cas que la funció generada yylex sigui un analitzador lexicogràfic.
- Des de la funció principal main (és l'opció que s'ha fet servir en aquest projecte).
- Des d'una altre funció.

Les tasques que es fan quan és crida a la funció yylex son:

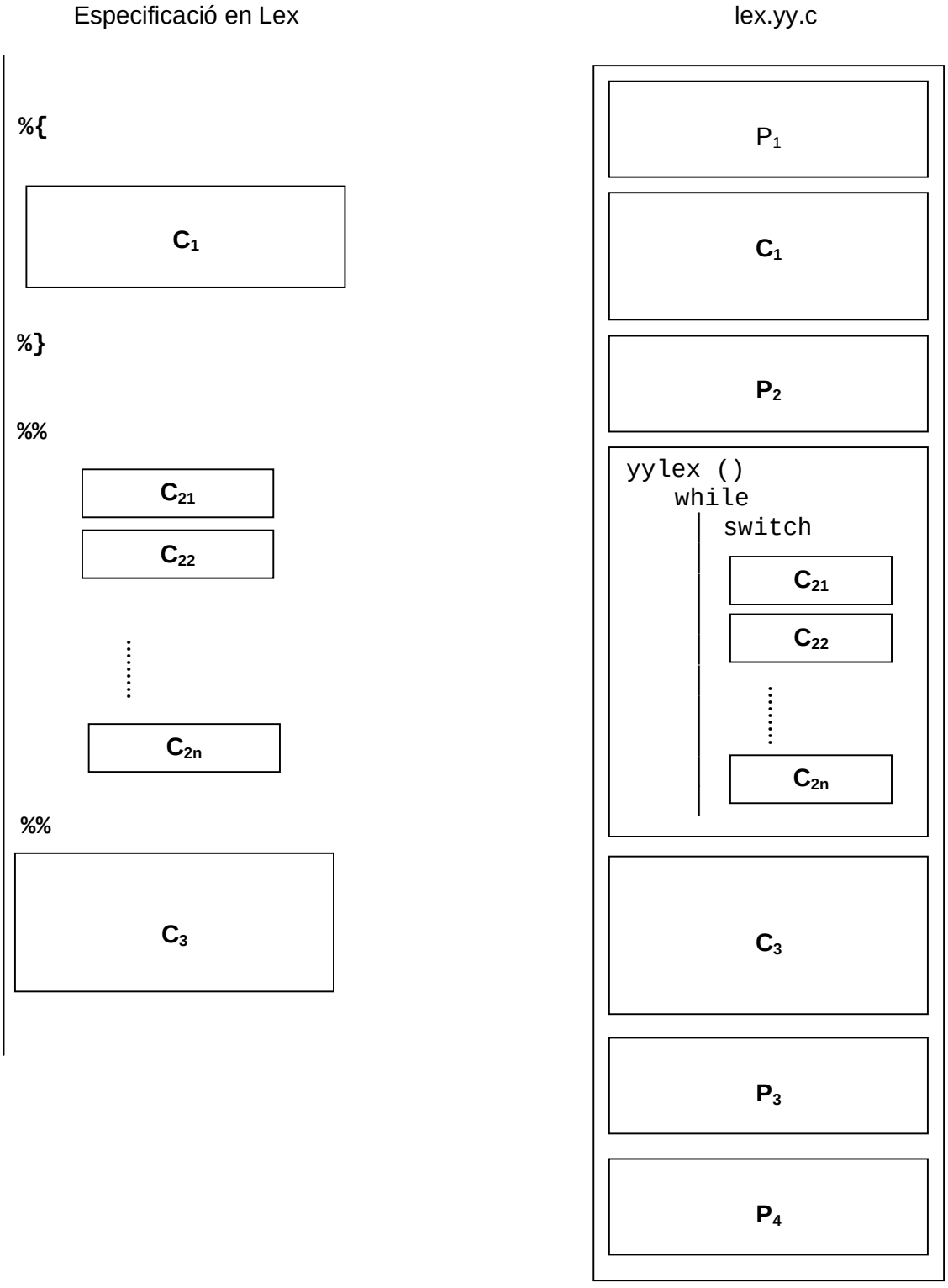
- Es llegeix tot el fitxer en una única cridada o cal realitzar diverses cridades per a completar la lectura.
- Es retorna algun valor o s'actualitzen variables globals.
- S'obtenen o no resultats relatius a tot el text gravat en el fitxer d'entrada.

4.2. Esquema General

Una especificació Lex està composta per 3 seccions, tal com es mostra en el següent esquema; la ratlla vertical de l'esquerra representa el començament de les línies del fitxer text d'entrada, és a dir la posició de la primera columna de cada línia.



En els paràgrafs següents s'explica la correspondència que és dóna entre les diferents parts de l'especificació d'entrada i l'estructura del fitxer de sortida; en aquestes explicacions és farà referència a la següent figura que resumeix la relació entre l'entrada i la sortida del traductor Lex (aquest esquema servirà per explicar amb exemples les diferents seccions).



Qualsevol de les 3 seccions d'una especificació Lex pot estar absent (encara que la secció de regles és la fonamental, i la seva absència fa que l'especificació resulti inútil).

El separador entre les seccions de definicions i de regles ha d'estar present, encara que les dues estiguin absents. El separador entre les seccions de regles i de rutines pot suprimir-se quan no hi ha secció de rutines. Dintre de la secció de definicions, qualsevol de les seves 2 parts pot estar present o no. Tots els separadors han de posar-se a partir de la primera columna.

Abans de prosseguir amb la descripció detallada de cadascuna de les seccions és mostrarà un exemple:

```
( 1) %{
( 2) int nPal = 0, nNum = 0;
( 3) %}
( 4)
( 5) lletra  [a-z]
( 6) xifra  [0-9]
( 7)
( 8) %%
( 9)
(10) {lletra}+   { ++nPal; }
(11) {xifra}+    { ++nNum; }
(12) \n          { ; }
(13) .           { ; }
(14)
(15) %%
(16)
(17) main () {


(18)   yylex ();
(19)   printf ("Paraules: %d\n", nPal);
(20)   printf ("Números : %d\n", nNum);
(21) }
```

- Línia (2): Es declaren els noms dels comptadors i se'ls assigna el valor inicial.
- Línia (5): S'indica que el nom lletra representa una lletra minúscula.
- Línia (6): S'indica que el nom xifra representa una xifra decimal.

- Línia (10): Es posa l'expressió regular que especifica una paraula i se li associa l'acció d'incrementar el comptador de paraules.

- Línia (11): Es posa l'expressió regular que especifica un nombre i se li associa l'acció d'incrementar el comptador de nombres.

- Línia (18): Es produeix la crida a la funció `yylex` que efectua l'anàlisi del text d'entrada.

- Línies (19) i (20): Es graven els valors finals dels comptadors.

4.2.1. Secció de definicions. Codi de definicions

Entre els delimitadors '%{' i '%}' es posa codi escrit en C que el traductor Lex trasllada al fitxer generat; aquest trasllat és literal i no es fa comprovació alguna; la correcció la verificarà el compilador de C . Es diu codi de definicions perquè és habitual que estigui format per definicions d'objectes als quals fa referència en el programa generat.

El codi es trasllada a la part del principi del programa generat, pel que queda col·locat abans que el codi de les accions associades als patrons; així doncs els noms definits aquí poden ser referits des del codi de les accions.

En el codi de definicions sol posar-se:

- Definicions de constants.
- Declaracions externes (globals).
- Declaracions #include (per a fitxers amb declaracions).

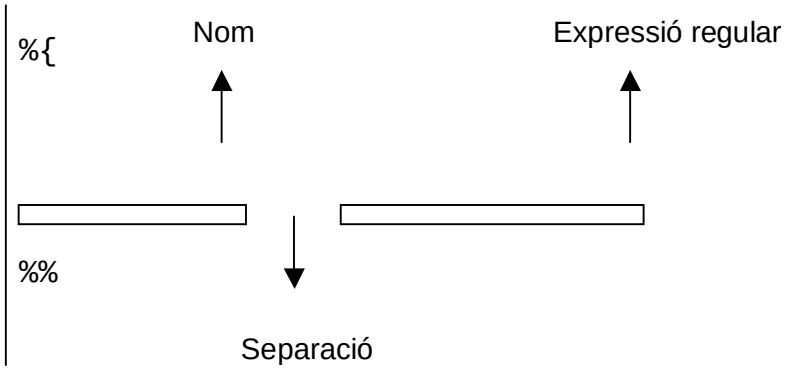
En general, es col·loca aquí el codi que degui quedar al principi del programa generat escrit en C.

En l'esquema comparatiu entre l'entrada i la sortida és la part representada per C_1.

4.2.2. Secció de definicions. Definicions regulars

Aquesta part de l'especificació serveix per a associar noms a expressions regulars; una vegada feta una associació, es pot incorporar el nom definit per a l'escriptura de noves expressions regulars. Així s'aconsegueix que les expressions regulars es puguin escriure de manera més llegible i còmoda.

L'associació entre un nom i una expressió regular es fa escrivint les dues en la mateixa línia, en primer lloc el nom i darrere l'expressió regular, separats per almenys un espai en blanc (o tabulador). El nom ha de posar-se a partir de la primera columna de la línia. Els noms es construeixen segons les mateixes normes dels identificadors en el llenguatge C (amb lletres, dígits i el caràcter guió inferior; no poden començar per un dígit).



Una vegada definit un nom, es pot usar immediatament; és a dir, es pot incorporar a l'escriptura de noves definicions regulars que es posin a continuació i també en les expressions regulars de la secció de regles.

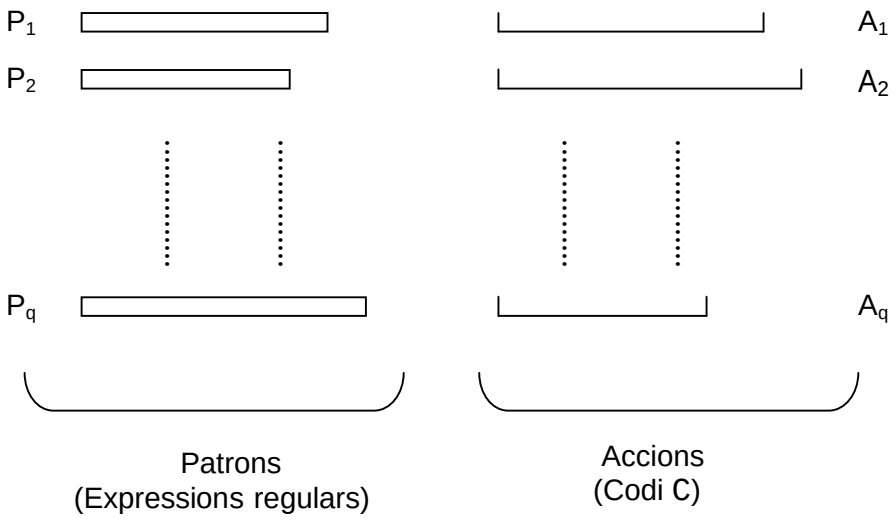
Si a l'escriure una expressió regular es vol emprar un nom prèviament definit, ha de posar-se aquest nom delimitat per claus, tal com es fa en les subexpressions.

Es mostrarà un exemple per la seva total comprensió:

```
%{

. . . . .

%}

vocal   (a|e|i|o|u)
lletraMin   [a-z]
paraulaMin  {lletraMin}+

%%

{vocal}[0-9]
{paraulaMin};
```

4.2.3. Secció de regles

Està formada per una seqüència de parells associats: patró, acció; el patró és una expressió regular i l'acció és un bloc de codi escrit en C .



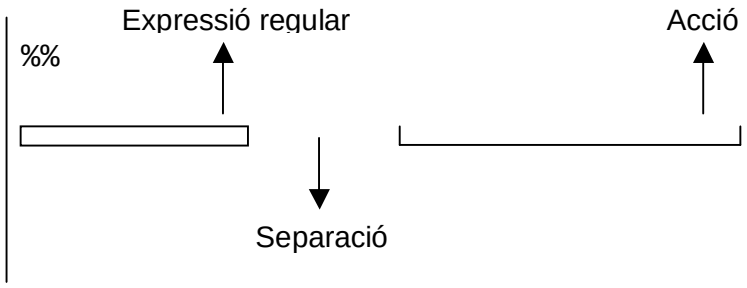
El que es pretén amb l'especificació d'aquests parells és que quan en la part del text d'entrada que en cada moment correspongui llegir, és troba una seqüència de caràcters que s'acobla a l'expressió regular d'un dels patrons especificats, llavors es procedeix a la realització de les operacions indicades pel codi de l'acció associada; també es sobrepassa (es considera ja llegida) la seqüència de caràcters acoplada.

La forma d'escriure les expressions regulars i el codi C ha de sotmetre's a certes normes:

- L'expressió regular de cada regla comença en la primera columna de la línia.

- Entre l'expressió regular i el començament del codi de l'acció deu haver-hi almenys un espai en blanc (o tabulador).

- Si el codi d'una acció té més d'una sentència o si s'estén al llarg de diverses línies, cal que vagi delimitat mitjançant claus; així doncs, les claus delimitadores poden evitar-se quan l'acció està formada per una única instrucció escrita en la mateixa línia on s'ha posat l'expressió regular; no obstant sol recomanar-se posar sempre les claus delimitadores per a afavorir la legibilitat de l'especificació.



L'acció associada a un patró mai pot estar buida; tota expressió regular ha de dur aparellada una acció. Si l'acció que correspon és no fer res, ha de posar-se el codi C que representa aquesta operació: la sentència buida (un simple punt i coma).

En l'esquema que relaciona l'entrada i la sortida del traductor Lex, el codi de les accions està representat per C_{21}, C_{22}, ..., C_{2n}; aquestes parts de codi especificades en l'entrada es traslladen literalment al codi de sortida; queden col·locades en el cos de la funció yylex, dintre d'una sentencia switch que selecciona l'acció corresponent al patró acoblat.

4.2.4. Secció de rutines

En aquesta secció s'inclou codi escrit en C que es trasllada literalment al fitxer generat; el que sol posar-se és el codi de les funcions a les quals s'ha fet referència des del codi de les accions de la secció de regles. Aquí no cal posar cura en la col·locació del codi en les línies i columnes, es traslladarà mantenint la col·locació i serà un text d'entrada per al compilador de C .

En l'esquema que relaciona l'entrada i la sortida del traductor Lex, el codi d'aquesta secció està representat per C_3.

4.2.5. Criteri de selecció

En la secció de regles es posa, com ja s'ha descrit, la relació dels patrons i les accions a ells associades.

Per a determinar el patró que es tria, en cada moment, per a acoblar una seqüència de caràcters de l'entrada cal aplicar el criteri:

1) A partir del caràcter actual s'intenten aplicar de manera simultània tots els patrons de l'especificació; hi haurà patrons que es puguin acoblar a la part actual de l'entrada, i uns altres que no es puguin.

2) De tots els patrons que s'acoblen a la part actual de l'entrada, es selecciona el que s'acopla a la seqüència de caràcters de major longitud (a partir del caràcter actual).

3) Si ocorre que diversos patrons s'acoblen sobre la mateixa seqüència de longitud màxima, es selecciona d'entre ells el que està situat abans en la relació de patrons de la secció de regles de l'especificació.

Segons aquest criteri, ocorre que, en general, no és indiferent l'ordre de col·locació dels patrons en la secció de regles.

Si, per exemple, en una especificació de regles es tenen, en l'ordre indicat a continuació, els patrons representatius d'una constant sencera i d'una constant decimal

```
|. . . . .
|. . . . .
|[0-9]+            { return pCteEnt; }
|[0-9]+\.[0-9]+    { return pCteDec; }
|. . . . .
|. . . . .
```

i en l'entrada la seqüència actual és 47.015+···, s'acoblarà la seqüència de longitud 6 que representa la constant decimal.

Cal tenir la cura de col·locar el patró dels identificadors darrere de tots els patrons de les paraules reservades; així, la col·locació dels patrons tindria que ser:

```
|. . . . .
|[eE][nN][dD]           { return prEnd; }
|. . . . .
|[wW][hH][iI][lL][eE]   { return prWhile; }
|. . . . .
|[a-zA-Z][a-zA-Z0-9]*   { return pId; }
|. . . . .
```

d'aquesta manera, quan en l'entrada es trobés la seqüència de caràcters ‘`end;`’, es considerarà com paraula reservada i no com identificador.

4.2.6. Emmagatzemament de la seqüència acoblada

El codi generat per `Lex` a partir de l'especificació d'entrada realitza la tasca de deixar anotada la seqüència de caràcters a la qual s'acobla el patró seleccionat.

Per a aquest emmagatzemament es disposa de dues variables globals, els noms de les quals són:

`yytext` conté la seqüència de caràcters acoblada
`yyleng` conté la longitud de la seqüència acoblada

Com ocorre amb l'emmagatzemament de les cadenes en el llenguatge `C`, el final de la seqüència de caràcters anotada s'indica mitjançant el caràcter especial de final de cadena.

El contingut de les variables `yytext` i `yyleng` canvia cada vegada que s'acobla (reconeix) una nova seqüència definida per un determinat patró; per això, si la seqüència actual es necessita per alguna operació posterior, haurà de traslladar-se a un altre lloc abans que s'acobli un nou patró.

Les tasques que es realitzen reiteradament per a analitzar el text d'entrada són les indicades en el següent esquema:

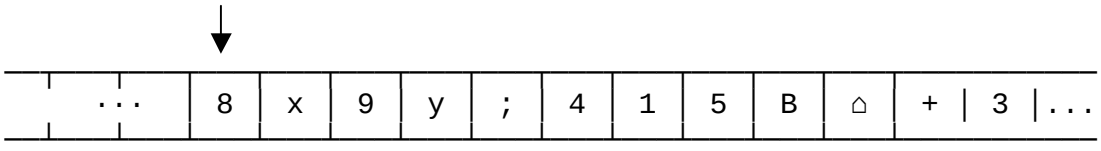
Mentre quedi text per analitzar:

- Acoblar un patró a partir del caràcter actual, aplicant el criteri de selecció de patrons descrit.
- Anotar la seqüència acoblada i la seva longitud en les variables globals *yytext* i *yyleng.*
- Avançar en el text d'entrada fins a arribar a sobrepassar la seqüència acoblada, que ja es considera analitzada.
- Executar l'acció associada al patró seleccionat.

A continuació es descriu un exemple en el qual es detallen les tasques que realitza l'algoritme produït per `Lex`. Siguin les regles:

```
        |  %%
P1      |  [;,]          { . . }          A1
P2      |  [0-9]+        { . . . . . . }  A2
P3      |  [a-z0-9]+     { . . . . }      A3
        |  %%
```

i sigui la situació en el fitxer d'entrada:



Els passos successius que realitza el programa generat són:

Pas i)

1.- Acoblament dels patrons i selecció:

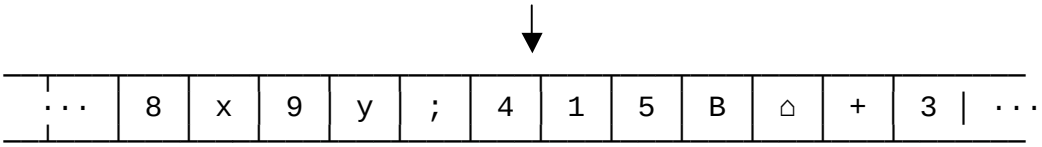
- P_1 no s'acobla
- P_2 s'acobla a la seqüència 8
- P_3 s'acobla a la seqüència 8x9i
- Es selecciona P_3.

2.- Emmagatzematge

`yyleng: 4`

	0	1	2	3	4	
yytext:	8	x	9	y	\0	

3.- Avanç en l'entrada:



4.- Execució de l'acció A3

Pas i + 1)

1.- Acoblament dels patrons i selecció:

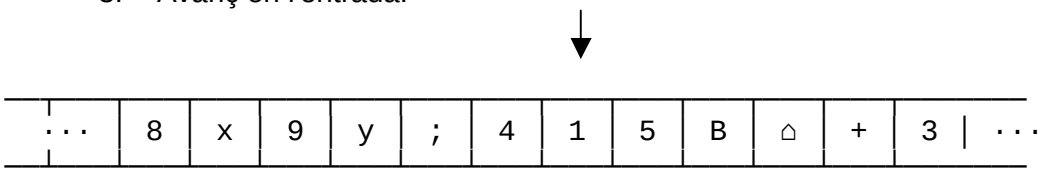
- P_1 s'acobla a la seqüència ;
- P_2 no s'acobla
- P_3 no s'acobla
- Se selecciona P_1.

2.- Emmagatzematge

`yyleng: 1`

	0	1	
yytext:	;	\0	

3.- Avanç en l'entrada:



4.- Execució de l'acció A1

Pas i + 2)

1.- Acoblament dels patrons i selecció:

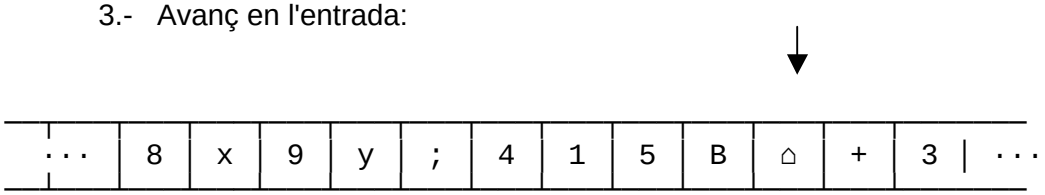
- P_1 no s'acobla.
- P_2 s'acobla a la seqüència 415.
- P_3 s'acobla a la seqüència 415.
- Se selecciona P_2.

2.- Emmagatzematge

`yyleng`: 3

	0	1	2	3	
`yytext`:	4	1	5	\0	

3.- Avanç en l'entrada:



4.- Execució de l'acció A2

Si cap de les accions A1 , A 2 i A 3 duu amb ella la devolució del control a la rutina que ha cridat a `yylex`, totes les operacions descrites en els tres passos anteriors es realitzaran dintre de la mateixa cridada a `yylex`; en cas contrari, es realitzarien en successives cridades a `yylex`.

4.2.7. Patró per defecte. Especificació completa

Si ocorre que el caràcter actual de l'entrada no s'acobla mitjançant cap dels patrons especificats en la secció de regles, llavors es considera que aquest caràcter queda acoblat amb el qual es pot denominar patró per defecte.

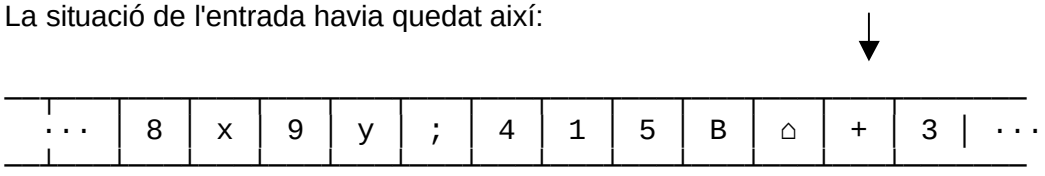
El patró per defecte té associada una acció: l'acció per defecte; aquesta acció consisteix a gravar en el fitxer de sortida predefinit el caràcter acoplat per defecte (sempre es tractarà d'un únic caràcter).

La selecció del patró per defecte i l'execució de l'acció associada a ell es realitzen seguint els mateixos passos que els descrits per als patrons de la secció de regles.

Atès que existeix el patró i l'acció per defecte, pot dir-se que l'analitzador generat per Lex mai es queda parat perquè no pugui acoblar cap patró a una entrada: en totes les situacions no especificades s'aplica el patró i l'acció per defecte.

Per a il·lustrar aquests conceptes es reprèn l'exemple anterior dedicat als passos que realitza el programa generat. Ara ja es pot apreciar que l'especificació de patrons en aquest exemple és incompleta.

La situació de l'entrada havia quedat així:



L'algoritme continua amb l'execució dels següents passos:

Pas i + 3)

1.- Acoblament dels patrons i selecció:

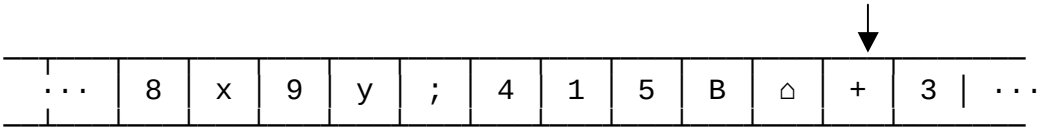
- P_1 no s'acobla
- P_2 no s'acobla
- P_3 no s'acobla
- Es selecciona el patró per defecte.

2.- Emmagatzematge

yyleng: 1

	0	1	
yytext:	B	\0	

3.- Avanç en l'entrada:



4.- Execució de l'acció associada al patró per defecte: es grava el caràcter B en el fitxer de sortida.

Pas i + 4)

1.- Acoblament dels patrons i selecció:

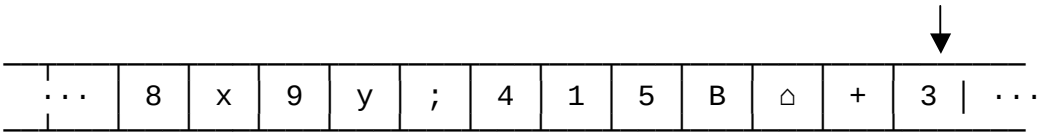
- P_1 no s'acobla
- P_2 no s'acobla
- P_3 no s'acobla
- Se selecciona el patró per defecte.

2.- Emmagatzematge

yyleng: 1

	0	1	
yytext:	\n	\0	

3.- Avanç en l'entrada:



4.- Execució de l'acció associada al patró per defecte: es grava un salt de línia en el fitxer de sortida.

Pas i + 5)

1.- Acoblament dels patrons i selecció:

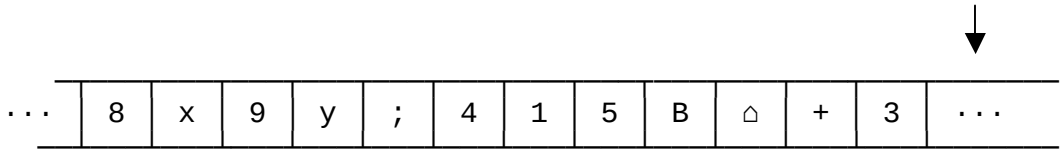
- P_1 no s'acobla
- P_2 no s'acobla
- P_3 no s'acobla
- Se selecciona el patró per defecte.

2.- Emmagatzematge

`yyleng`: 1

	0	1	
`yytext`:	+	\0	

3.- Avanç en l'entrada:



4.- Execució de l'acció associada al patró per defecte: es grava el caràcter + en el fitxer de sortida.

Si, en aquest exemple, per a aconseguir una especificació completa es posa:

```
        %%
P1      [;,]            { . . }           A1
P2      [0-9]+          { . . . . . . }   A2
P3      [a-z0-9]+       { . . . . }       A3
        .|\n            { ; }
        %%
```

s'haurien executat les mateixes operacions indicades en els passos *i + 3*, *i + 4* i *i + 5*, excepte l'enregistrament dels tres caràcters en el fitxer de sortida.

4.3. Obtenció i execució de l'analitzador

L'analitzador lèxic s'obté després de la realització dels següents passos:

1) **`vi eLexic.l`** Edició del fitxer amb les característiques lexicogràfiques
2) **`lex eLexic.l`** Traducció de les característiques lexicogràfiques
3) **`cc lex.yy.c -o Traductor -ll`** Compilació del codi

L'ordre de passos citat és una possibilitat, no una necessitat; en aquest projecte s'ha fet ús de l'editor.

Els fitxers sobre els quals actua l'analitzador lèxic generat són (llevat que de manera explícita s'indiqui una altra cosa) els predefinits d'entrada i de sortida; així doncs, l'execució de l'analitzador obtingut pot fer-se d'una les següents formes:

Traductor

Traductor <Entrada

Traductor <Entrada >Sortida

segons que es faci o no redireccionament dels fitxers d'entrada i de sortida predefinits.

Al fitxer d'entrada es proporciona un programa escrit en L. La sortida és l'arxiu traduït al codi `C`, si no s'indica arxiu de sortida es mostrarà el codi per pantalla.

4.4. Expressions regulars

Es farà una petita explicació les expressions regulars, si es vol més informació dels caràcters que es poden utilitzar en `Lex` i totes les expressions possibles consultar els llibres descrits a l'apartat de bibliografia.

Les expressions regulars tal com s'estudien en la teoria de llenguatges per a especificar els llenguatges regulars, estan constituïdes per símbols d'un alfabet Σ relacionats mitjançant els operadors binaris alternativa (**|**) i concatenació (·) i l'operador estrella (*****); en l'escriptura d'una expressió regular també es poden emprar parèntesi per a precisar l'ordre d'aplicació dels operadors. L'asterisc de l'operació estrella sol col·locar-se com exponent de la part de l'expressió regular afectada.

La precedència dels operadors és la definida per la següent jerarquia, relacionada de major a menor precedència:

1 Operacions entre parèntesis
2 Operador estrella
3 Operador concatenació
4 Operador alternativa

Així, per exemple, són expressions regulars definides sobre l'alfabet $\Sigma = \{a,b\}$

$$b \cdot a \cdot a | b \cdot b$$

$$a^{*} \cdot (b | b \cdot a)$$

La primera denota el llenguatge regular format per dues paraules $\{baa, bb\}$ i la segona denota el llenguatge regular d'infinites paraules $\{b,\ ba,\ ab,\ aba,\ aab,\ aaba,\ \cdot\cdot\}$.

Entre els símbols que apareixen en una expressió regular cal distingir els caràcters i els metacaràcters; els caràcters són els símbols que pertanyen a l'alfabet sobre el qual està definida l'expressió regular; els metacaràcters són els símbols que no pertanyen a aquest alfabet: els operadors i els parèntesis.

En l'escriptura de les expressions regulars el punt representatiu de la concatenació entre símbols de l'alfabet sol suprimir-se; d'acord amb aquesta notació simplificada, les anteriors expressions solen escriure's així:

$$baa | bb$$

$$a^{*} (b | ba)$$

Ja que l'espai en blanc no és un símbol pertanyent a l'alfabet Σ sobre el qual estan definides les expressions regulars anteriors, també podrien escriure's d'aquesta manera:

```
baa | bb
a*  ( b | ba )
```

En una especificació Lex s'inclouen expressions regulars, però escrites amb una notació que és una ampliació de la notació emprada en la definició anterior. Aquesta ampliació té com principals objectius:

- Fer més còmoda i curta l'escriptura de les expressions regulars.
- Distingir de manera precisa els caràcters de l'alfabet i els metacaràcters emprats en l'escriptura de les expressions regulars.

Les expressions regulars emprades en les especificacions Lex no tenen conceptualment cap diferència amb les expressions regulars que defineixen els llenguatges formals regulares; l'únic que aporten són modificacions en la notació emprada en l'escriptura de les expressions en forma de seqüències de caràcters consecutius susceptibles de gravar-se en un fitxer de tipus text. Aquesta notació ampliada aconsegueix certa dificultat per les següents causes:

- Per a facilitar i fer mes curta l'escriptura de les expressions s'introdueixen bastants metacaràcters que s'intercalen amb els caràcters.

- És habitual que qualsevol caràcter de l'alfabet ASCII formi part de l'alfabet sobre el qual es defineixen les expressions regulars i que, per tant, pugui aparèixer en elles (fins i tot l'espai en blanc, el tabulador o la final de línia).

A continuació s'utilitzarà la notació ⟶ per indicar que es posa a continuació la descripció del conjunt de paraules denotades per l'expressió regular que els precedeix. Els exemples que acompanyen a les descripcions es refereixen a l'alfabet Σ = {x, y, z}.

A més de les operacions emprades en les expressions regulars (alternativa, concatenació i estrella, amb la jerarquia entre elles que abans s'ha citat), en les expressions regulars Lex s'incorporen les operacions que es descriuen a continuació.

4.4.1. Operació opcionalitat:?

Representa la presència d'una única vegada o l'absència de les paraules denotades per l'expressió regular afectada; té la mateixa prioritat que l'operació estrella. En general, si α és una expressió regular, es verifica l'equivalència entre α? y α|ε. Per exemple,

```
x|y?      ⟶  {x, y, ε}
xz?|zzy   ⟶  {x, xz, zzy}
```

4.4.2. Operació repetició una o més vegades: +

Representa la presència d'una o més vegades consecutives de les paraules denotades per l'expressió regular afectada; té la mateixa prioritat que l'operació estrella. En general, si α és una expressió regular, es verifica l'equivalència entre α+ y αα*. Per exemple,

```
yx+      ⟶  {yx, yxx, yxxx, · · ·}
(x|y)+z  ⟶  {xz, yz, xxz, xyz, yxz, yyz, · · ·}
```

4.4.3. Operació repetició limitada: { }

Representa la presència un nombre donat i limitat de vegades consecutives de les paraules denotades per l'expressió regular afectada; té més prioritat que l'alternativa però menys que la concatenació.

La quantitat de presències s'indica mitjançant els nombres posats entre les claus; hi ha tres maneres de fer-ho:

`{n}` presencia de n vegades

`{n,m}` presencia de i vegades, `n ≤ i ≤ m`

`{n,}` presencia de i vegades, `n ≤ i`

4.4.4. Corxets: [ ]

Per a indicar, dintre d'una expressió regular Lex, la presència d'un caràcter de l'alfabet triat entre un conjunt de caràcters, s'introdueix una notació més còmoda que empra els corxets com metacaràcters; en el lloc de l'expressió regular on hagi d'estar el caràcter es posa entre corxets el conjunt dels caràcters possibles; tots els caràcters del conjunt es posen consecutius, sense separació alguna entre ells.

La descripció dintre dels corxets dels caràcters que pertanyen al conjunt pot fer-se de varies maneres; són les següents:

a) Descripció per enumeració: `[aeiou]`
b) Descripció per enumeració complementària: `[^UOIEA]`
c) Descripció per rang: [0-9]
d) Descripció combinada: `[a-zA-Z]`

En totes les maneres de descripció és indiferent l'ordre de col·locació dels caràcters i també pot haver-hi repeticions (que resulten inútils).

Una vegada descrits els nous operadors afegits a les expressions regulars Lex, es farà ara una recapitulació dels operadors disponibles i de la jerarquia de la seva aplicació; relacionada de major a menor, és la següent:

1 Operacions entre parèntesis
2 Operadors **+, * i ?**
3 Operació de concatenació (absència d'operador)
4 Operació de repetició limitada **{ }**
5 Operador **|**
6 Operador [ ]

Els operadors que tenen la mateixa prioritat s'apliquen d'esquerra a dreta, segons es troben en l'expressió regular; els operador `*, +, ?, {, }` s'apliquen a la part de l'expressió regular que està situada a la seva esquerra.

A continuació es mostraran uns exemples de escriptura en `Lex`:

`(n|N)(o|O)|(s|S)(i|I)` → paraules *no o si* amb lletres minúscules o majúscules .

`[nN][oO]|[sS][iI]` → mateixes vuit paraules que l'expressió anterior.

`3[a-z]|[A-Z]*9` → paraules la forma de les quals és un 3 seguit d'una lletra minúscula o bé zero o més lletres majúscules seguides d'un 9.

`[xyz]+` → paraules de longitud major o igual que 1, formades amb les lletres x, i, *z.

4.4.5. Metacaracters \ i “

En les expressions regulars `Lex` tenim dues maneres de suprimir la qualitat de metacaràcter:

1 Mitjançant la contrabarra \
2 Mitjançant les cometes "

Així doncs, el símbol ‘\’ és un nou metacaràcter, tant extern com intern, ja que en qualsevol dels dos contextos en els quals es trobi no se li considera amb el seu significat pròpiament dit. A continuació s'exposen uns exemples.

`\+|-` → un dels dos operadors aritmètics additius

`[01\-63]` → `{0, 1, 3, 6, -}`

Així doncs, el símbol '"' és un nou metacaràcter extern. A continuació s'exposen uns exemples.

"Abans" ⟶ la paraula (de cinc caràcters) Abans

"+""-""*""/" ⟶ la paraula (de quatre caràcters) +-*/

4.4.6. Seqüències d'escape

Totes les seqüències d'escape del llenguatge de programació C s'admeten en les expressions regulares Lex amb la mateixa notació i el mateix significat :

```
\n      \t      \v      \b      \r      \a      \f
```

Les dues primeres refereixen al final de línia i al tabulador (horitzontal). En totes elles la presència del símbol '\' no es considera amb el significat de metacaràcter que té en altres situacions; això és, la lletra que constitueix el segon caràcter de la seqüència no representa la lletra pròpiament dita.

4.4.7. El caràcter universal

En les expressions regulars Lex el símbol punt ('.') és un metacaràcter que representa un caràcter qualsevol de l'alfabet sobre el qual estan definides les expressions, excepte el caràcter representatiu del final de línia; es tracta d'un metacaràcter extern. Per tant quan s'empra el símbol punt com caràcter s'ha de posar una de les notacions que suprimeixen la seva qualitat de metacaràcter. Així en:

`a[.:]b` ⟶ {a.b, a:b}

`a(\.|:)b` ⟶ les mateixes paraules que l'expressió anterior

4.4.8. Subexpressions amb nom

Per a fer més còmoda o més àgil l'escriptura de les expressions regulars Lex és possible associar un nom a una expressió regular i, després, usar aquest nom com subexpressió d'una expressió regular més complexa.

Per a utilitzar com a component d'una expressió regular el nom d'una altra expressió definida prèviament n'hi ha prou amb incorporar aquest nom delimitat per claus en el lloc en el qual procedeixi col·locar la subexpressió.

Per exemple, en cadascuna de les següents línies s'associa un nom a una expressió regular:

```
xifres       [0-9]+
lletrasMin    [a-z]{1,3}
lletraMay     [A-Z]
Asignacio   :=
puntYcoma   ;
```

Una vegada definits aquests noms, es poden emprar com components d'altres expressions regulars; així per exemple:

```
1) x{xifres}
2) {xifres}{lletrasMin}
3) \({lletraMay}*\)
4) a{Asignacio}{lletrasMin}{puntYcoma}
```

Les quatre expressions regulars abans definides són:

```
1) x[0-9]+
2) [0-9]+[a-z]{1,3}
3) \([A-Z]*\)
4) a:=[a-z]{1,3};
```

S'ha de prestar especial atenció al definir els noms de les subexpressions per evitar resultats erronis al fer-se la substitució literal. El següent exemple mostra una substitució que produeix resultats habitualment inesperats. Siguin les definicions:

```
consonant  t|d
vocal  a|e|i|o|u
```

si s'escriu l'expressió regular `{consonant}{vocal}`, és pot pensar que s'estan especificant les paraules formades per una consonant seguida d'una vocal; però no és així, la substitució literal indica que l'expressió regular realment escrita és:

```
t|da|e|i|o|u
```

Per a obtenir el resultat suposadament esperat caldria definir els noms així:

```
consonant (t|d)
vocal (a|i|i|o|o)
```

4.4.9. Relació de metacaràcters

A continuació es posa una relació resumida dels metacaràcteres, externs i interns, de les expressions regulars Lex que s'han descrit en aquesta memòria.

- Externs:

.	Representació de qualsevol caràcter, excepte el final de línia.
(·····**)**	Modificació en l'ordre d'aplicació de les operacions.
|	Operació alternativa.
*****	Operació estrella.
?	Operació opcionalitat.
+	Operació repetició una o més vegades.
{·····**}**	Operació repetició limitada: quan la seqüència entre claus comença per un dígit; els caràcters que estan entre les claus formen part de l'operador.
{·····**}**	Incorporació del nom d'una subexpressió: quan la seqüència entre claus comença per una lletra o pel caràcter guió inferior; els caràcters que estan entre les claus formen el nom i també són metacaràcters.
[·····**]**	Definició d'un conjunt de caràcters.
\	Supressió de la qualitat de metacaràcter; excepte en les seqüències d'escape.
"....."	Supressió de la qualitat de metacaràcter; excepte per al caràcter '\'.
^	Ajust al principi d'una línia; quan és el primer símbol de l'expressió.

$ Ajust al final d'una línia; quan és l'últim símbol de l'expressió.

/ Fixació d'un context per la dreta.

<·····> Condició de context; quan està davant del primer símbol de la expressió.

` ` Representa en aquesta relació el caràcter espai en blanc; l'espai en blanc com metacaràcter significa la terminació de l'escriptura d'una expressió regular.

- Interns:

^ Definició complementària d'un conjunt.

- Rang de caràcters en un conjunt.

**** Supressió de la qualitat de metacaràcter; excepte en les seqüències d'escape.

4.5. Definició de les paraules reservades del projecte

Una vegada s'ha explicat que és Lex i com aquesta eina ens pot ajudar a fer un traductor, de quina manera funciona, com troba les paraules reservades, quines accions fa, com es compila, en quin llenguatge es programa i quines són les seves expressions regulars, ara s'explicaran les paraules reservades que s'han utilitzat per generar la traducció del pseudocodi. Cadascuna d'aquestes paraules te diferents necessitats depenent de la seva traducció (obrir claus, posar ';' , tancar corxets, etc..) i per els casos mes complicats s'explicarà de quina manera s'han solucionat aquestes necessitats.

Per poder facilitar l'explicació de les diferents paraules reservades agruparem aquestes en diferents grups:

4.5.1. Paraules amb traducció directe

Son aquelles paraules reservades que la seva traducció és directe, no influeixen en la traducció d'altres i no necessiten cap altre element perquè siguin vàlides (exceptuant el ; que ja s'explicarà més endavant).

Aquestes paraules sempre s'emmagatzemen per saber l'ultima paraula traduïda.

[a-z]

- Serà reconeguda pel patró quan sigui qualsevol lletra que no hagi estat reconeguda pels altres patrons.

[0-9]

- Serà reconeguda pel patró quan sigui un número del `0-9`.

CONST

- La seva traducció serà '`const`'.

ALTRAMENT

- La seva traducció serà '`default`'.

FI_SI

- La seva traducció serà '}' , el tancament del `if`.

FI_SEGONS

- La seva traducció serà '}' , el tancament del `switch`.

FI_MENTRE

- La seva traducció serà '}' , el tancament del `while`.

FI_PER | FI_DES_DE

- La seva traducció serà '}', el tancament del `for`.

FI_ESTRUCTURA

- La seva traducció serà '`};`' , el tancament de `struct`.

GLOBAL | FI_GLOBAL

- Es tradueixen per paraula buida (`''`) ja que només indiquen la secció de les declaracions de les variables, en C aquesta secció no s'ha d'indicar.

FINAL

- La seva traducció serà '}', el tancament del `main`.

FI_FUNCIO

- La seva traducció serà '}', el tancament d'una funció.

SURT | SORTIR

- La seva traducció serà '`return;`'.

ABANDONA

- La seva traducció serà '`break;`'.

CONTINUA

- La seva traducció serà '`continue;`'.

RETORNA

- La seva traducció serà ‘`return`’.

DIV

- La seva traducció serà ‘`/`’.

MOD

- La seva traducció serà ‘`%`’.

O

- La seva traducció serà ‘`||`’.

I

- La seva traducció serà ‘`&&`’.

NO

- La seva traducció serà ‘`!`’.

4.5.2. Paraules amb traducció directe però que la seva presencia és important

Son aquelles paraules reservades en les que la seva traducció és directe però que la seva presencia implica la modificació d’altres paraules reservades o que necessiten altres elements perquè la seva estructura sigui correcte (obrir parèntesis, tancar claus, etc..).

ENTER

- La seva traducció serà ‘`int`’.
- Abans d’escriure la traducció al text de sortida s’emmagatzemarà en una variable el tipus declarat ja que totes les paraules que s’escriguin a continuació i fins que s’acabi la línia seran del tipus `int` i es tindran que guardar en un array perquè potser més tard necessitem consultar el tipus de variable per la traducció correcte d’altres paraules (com és el cas del `printf` i l’`scanf`).

REAL

- La seva traducció serà '`double`'.
- Abans d'escriure la traducció al text de sortida s'emmagatzemarà en una variable el tipus declarat ja que totes les paraules que s'escriguin a continuació i fins que s'acabi la línia seran del tipus `double` i es tindran que guardar en un array perquè potser més tard necessitem consultar el tipus de variable per la traducció correcte d'altres paraules

CAR

- La seva traducció serà '`char`'.
- Abans d'escriure la traducció al text de sortida s'emmagatzemarà en una variable el tipus declarat ja que totes les paraules que s'escriguin a continuació i fins que s'acabi la línia seran del tipus `char` i es tindran que guardar en un array perquè potser més tard necessitem consultar el tipus de variable per la traducció correcte d'altres paraules.

SI

- La seva traducció serà '`if (`'.
- Aquesta paraula implica que tindrà que tancar-se un parèntesis al finalitzar línia i obrir corxets en el cas en que hi hagi més d'una línia de codi a realitzar si es compleix la validació.

LLAVORS

- La seva traducció serà ') {'.
- Aquesta paraula implica que tindrà que tancar-se una clau al finalitzar l'estructura del `SI` o al trobar un `SI_NO`.

SI_NO

- La seva traducció serà '`else`'.
- Abans d'escriure la traducció al fitxer de sortida hem de mirar si l'`if` que s'ha escrit anteriorment portava claus ja que d'aquesta manera la traducció seria '`} else {`' i es tindria que tancar clau després de finalitzar la condició de l'`else`.

SEGONS

- La seva traducció serà '`switch (`'.
- Després d'escriure l'expressió que avaluarà el `switch` s'ha de tancar el parèntesis que s'ha obert anteriorment a la declaració, a continuació obrir claus per englobar tot el codi del `switch` i quan aquest hagi finalitzat tancar claus.

```
switch (condicio){
      ....
}
```

CAS

- La seva traducció serà '`case`'.
- Després d'escriure al text de sortida totes les accions que es faran si l'expressió avaluada al `switch` es compleix, hem descriure '`break;`' per sortir del `switch`.

```
switch (condicio){
      case:
            .....
            break;
}
```

MENTRE

- La seva traducció serà '`while (`'.
- Després d'escriure l'expressió que avaluarà el `while` s'ha de tancar parèntesis.

FER

- La seva traducció serà '`){`'.
- Després d'escriure totes les accions que es generaran si es compleix la condició s'ha de tancar claus.

```
while (condicio) {
      ...
}
```

REPETIR

- La seva traducció serà '`do {`'.
- Després d'escriure totes les accions que es generaran si es compleix la condició s'ha de tancar claus.

ESTRUCTURA

- La seva traducció serà '`struct`'.
- Després descriure el nom de l'estructura es tindran que obrir claus i al finalitzar la declaració de les seves variables tancar claus.

```
struct nom {
     int ...
     float ..
}
```

ALLIBERA | ALLIBERAR

- La seva traducció serà '`free (`'.
- Després d'escriure la variable a alliberar es tindrà que tancar parèntesis.

INICI

- La seva traducció serà:

```
#include <stdio.h>
#include <stdlib.h>
#include "Funciones.h"
int main (int argc, char *argv[]) {
```

- Després d'escriure la traducció i tot el codi que engloba el `main` es tindrà que tancar parèntesis.

```
int main (int argc, char *argv[]) {
   ...
   ...
}
```

/*

- La seva traducció serà '`/*`'.
- Abans de fer la traducció s'ha de mirar si el final de línia necessita un punt i coma, un corxet, etc..

`*/`

- La seva traducció serà '`*/`'.
- En aquest cas mai hem de posar punt i coma ni altre final de línia.

`, (coma)`

- La seva traducció serà ','.
- Es important reconèixer aquesta paraula perquè si hem declarat un tipus de variable indica que la següent paraula que es reconeixerà serà una variable del tipus declarat i es tindrà que emmagatzemar en el seu corresponent array.

4.5.3. Paraules amb traducció complexa

Son aquelles paraules reservades que tenen una traducció mes complexa ja que aquesta no sempre és la mateixa o no és tan directa.

`LLEGEIX I ESCRIU`

En llenguatge C quan llegim o escrivim una variable hem d'indicar quin tipus de dades es:

. Integer: d
. Char: s
. Double: f

Per poder saber de quin tipus de dades és cada variable s'han creat tres arrays, quan el codi troba una paraula 'ENTER', 'REAL' o 'CAR' significa que ens preparem per declarar variables, llavors totes les paraules que vinguin darrera i fins que trobem un final de línia seran variables del tipus declarat i es guardaran en el seu corresponent array.

Exemple:

Pseudocodi:

```
ENTER aux, prm
CAR var
```

Forma de guardar al Lex:

```
Arrayint = [aux, prm]
arraychar = [var]
```

Un cop trobem un 'LLEGEIX' o 'ESCRIU' hem de llegir la variable que es vol escriure o llegir, mirar de quin tipus és comparant-la amb les variables dels 3 arrays, i per últim escollir la lletra que li correspon depenent del tipus de variable que sigui.

Exemple:

Pseudocodi:

```
ENTER aux, prm
CAR var
LLEGEIX aux
ESCRIU var
```

Després de cridar al Lex:

```
int aux, prm;
char var;
scanf("%d", &aux);
printf("%s", var);
```

FINS_QUE

- La seva traducció serà '`} while (`'.
- Després de l'expressió s'ha de tancar claus.
- Si ens fitxem el significat de FINS_QUE és totalment contrari que el del `while` (mentre), així que abans d'escriure l'expressió que s'avaluarà hem de negar-la '`!( )`'.

PER | DES_DE

- La seva traducció serà '`for (`'
- Després d'escriure tota la definició del `for` s'haurà de tancar parèntesis i obrir claus per començar a declarar les accions.
- Es tindrà que guardar en memòria la variable declarada ja que aquesta es necessitarà per indicar-la en els tres trams de la declaració.

```
for (i=0;i<10;i++) {
....
}
```

FINS_A

- La seva traducció serà ‘;’ més la variable declarada en el for i la condició aplicada posteriorment.

  Exemple:

  ```
  PER i=0 FINS_A 3   ⟶   for (i=0; i<=3;
  ```

INCR | INCREMENT| CADA

- La seva traducció serà ’;’ més la variable declarada en el for i el seu increment.
- Després d’escriure la condició d’increment es tindrà que tancar parèntesis i obrir clau.

  Exemple:

  ```
  PER i=0 FINS_A 3 INCRE +1
  for (i=0; i<=3; i=i+1) {
  ```

ALLOTJA | ALLOTJAR | RESERVA | RESERVAR

- La seva traducció serà diferent depenent de la declaració de ALLOTJA que es faci, aquesta port ser:

  ```
  ALLOTJA x
  ALLOTJA x, n
  ```

  La primera reserva memòria per x i la segona reserva n espais de memòria per x. La seva traducció en C serà respectivament:

  ```
  x=malloc(sizeof(*x));
  x=malloc(n*sizeof(*x));
  ```

- D’aquesta manera podem observar que s’ha d’emmagatzemar d’alguna manera les variables de l’allotja per poder crear la frase correcte en C, ja que aquestes s’utilitzen al llarg de tota la declaració.

INICI_FUNCIO

- La seva traducció serà diferent depenent de la frase escrita al text del `pseudocodi`, ja que la funció pot retornar una variable:

  . Si retorna una al text en `pseudocodi` s'ha informar de quin tipus es.

  . Si no retorna cap s'ha de que posar `'void'`.

  Per exemple:

  Si retorna un valor:

  Text en `pseudocodi`:

```
ENTER Intercanvia (ENTER *pm, ENTER *pn)
INICI_FUNCIO
...
FI FUNCIO
```

  La seva traducció a C seria:

```
int Intercanvia (int *pm, int *pn) {
     ...
}
```

  Si no retorna res:

  Text en `pseudocodi`:

```
Intercanvia (ENTER *pm, ENTER *pn)
INICI_FUNCIO
...
FI FUNCIO
```

  La seva traducció a C seria:

```
void Intercanvia (int *pm, int *pn) {
     ...
}
```

- Com es pot veure primer es declara la funció i després tenim la paraula reservada, això complica la traducció perquè fins ara sempre es va escrivint a mida que es van lleguin les paraules claus. Per això en aquest cas s'ha de guardar tota la frase i després escriure-la.

- També hem d'observar que a C les funcions s'han de declarar abans del main, com ja s'ha explicat, amb l'aplicació Lex escrivim a mida que llegim, per això quan estem llegint la funció no podem anar cap amunt a escriure la seva declaració, la solució que es va trobar a aquest problema és crear un nou arxiu anomenat Funcions.h i afegir aquí totes les definicions de funcions. Si al final de la traducció l'arxiu no te cap funció declarada aquest arxiu s'elimina.

Després d'explicar els diferents grups de paraules reservades és important destacar la importància del ';' al final de les línies de l'arxiu C perquè aquest sigui correcte. Com normalment aquest caràcter s'escriu sempre exceptuant determinats casos, no s'ha explicat si es posa o no a cadascuna de les paraules reservades exposades anteriorment, només es posaran les seves excepcions.

- Quan l'ultima paraula escrita sigui:

```
. }
. {
. ;
. else
. '' (res)
```

no s'escriurà ';'.

5. L'SCRIPT COMPILADOR

5.1. Definició

L'executable Lex creat només tradueix un arxiu escrit en pseudocodi en un arxiu escrit en C, però per facilitar el seu ús es vol crear un compilador de C. Per poder fer això s'ha utilitzat la comanda gcc i perquè tots els passos de traduir i compilar siguin molt mes senzills per els usuaris que l'utilitzen s'ha creat un script per bash que tradueix i compila l'arxiu escrit en pseudocodi.

Un script per bash és un arxiu tipus text on les línies tenen comandes que són executades per bash.

bash és un producte GNU (software de distribució lliure) descendent del *Bourne Shell* , és d'interfície estàndard de la línia de comandes en la majoria de maquines linux. Potencia l'interavilitat suportant l'edició en la línia de comandes, capacitat de completar automàticament una comanda, etc.

L'script que s'ha creat funciona com la comanda gcc, és a dir, reconeix les opcions que es passen per paràmetre a gcc:

- gcc –c a.c : Compila, crea a.o.
- gcc – o b b.c: Compila i linka, crea b.
- gcc –o b a.o b.c: Compila b i linka a.o i b.o, crea l'executable b.

En el nostre cas en compte d'escriure gcc cridarem a l'script anomenat 'compila' i per paràmetres podem posar arxius en pseudocodi (.psc), arxius en C (.c), arxius objectes (.o) o el nom de l'executable. Depenent de l'opció que es passi per paràmetre realitzarà unes funcions o unes altres:

–t) Tots el .psc de la línia de comandes es traduiran i donaran el .c corresponent. Els .c i .o s'ignoraran.

–c) Tots els .psc i .c es compilaran i donaran el .o. Els .psc abans es traduiran i els .o s'ignoraran.

Si no hi ha ni –c ni –t: Els .psc i .c donaran el .o corresponent i, juntament amb la resta de .o generaran un executable (si hi ha un –o ens indicarà el nom de d'executable, si no per defecte s'anomenarà a.out).

Es mostraran uns exemples de la seva funcionalitat:

- ./compila a.psc: Tradueix l'arxiu a.psc creant l'arxiu a.c i compila creant l'executable a.out.
- ./compila a.c: Compila creant l'executable a.out.
- ./compila –o a a.c: Compila creant l'executable a.
- ./compila –c a.psc: Tradueix l'arxiu a.psc creant l'arxiu a.c i el compila creant a.o.
- ./compila –c a.c: Compila creant a.o.
- ./compila –t a.psc: Tradueix l'arxiu a.psc creant l'arxiu a.c.

5.2. Comandes internes que s'han utilitzat

5.2.1. getopts

Permet processar opcions de línia de comandes més còmodament. A més permet processar opcions quan estan agrupades (-abc) i arguments d'opcions quan aquests no estan separats de la opció per un espai (-barg en comptes de –b arg).

Aquesta comanda s'ha utilitzat per poder processar d'una manera més senzilla les opcions –c, -t i –o.

5.2.2. sed

Es un editor no interactiu, sed substitueix la cadena a substituir per una altre cadena. El fitxer original no queda alterat, sed obliga a guardar els canvis en un altre fitxer.

Aquesta comanda s'ha utilitzat per canviar el nom dels arxius `.h` definits als arxius `.c` ja que no s'anomenaran `Funcions.h` si no `Nomarxiu.h` o per eliminar la línia en la que es defineix si l'arxiu `Funcions.h` és vuit.

5.2.3. `indent`

Permet establir el nivell de sangria, pot aplicar-se a elements del mateix bloc per definir la quantitat de sangria que tindria que aplicar-se a la primera línia.

Aquesta comanda s'ha utilitzat per automatitzar el procés d'estilització de l'arxiu `.c` creat.

5.2.4 `gcc`

`gcc` (GNU Compiler Collection) és una col·lecció de compiladors que permet diversos llenguatges com `C, C++ i Java.`

Aquesta comanda s'ha utilitzat per compilar i linkar els arxius .c per poder generar un arxiu executable.

6.- RESULTATS

A continuació es mostraran dos exemples d'arxius en pseudocodi traduïts amb l'eina creada en aquest projecte en arxius en C.

P1.psc

```
INICI
     ENTER n, *prm, i, j
     REPETIR
          LLEGEIX n
     FINS_QUE n > 1

     ALLOTJA prm, n+1

     PER i = 2 FINS_A n INCR +1
          prm[i] = 1
     FI_PER

     i = 2 /* es prescindeix de prm[0] i de prm[1] */

     MENTRE i * i <= n FER
          PER j = i FINS_A n DIV i INCR +1
               prm[j * i] = 0
          FI_PER
          REPETIR
               i = i + 1
          FINS_QUE prm[i]
     FI_MENTRE

     PER i = 2 FINS_A n INCR +1
          SI prm[i] LLAVORS
               ESCRIU i
          FI_SI
     FI_PER

     ALLIBERA prm
FINAL
```

P1.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int n, *prm, i, j;
    do {
     scanf("%d", &n);
    } while (!(n > 1));
    prm = malloc((n + 1) * sizeof(*prm));
    for (i = 2; i <= n; i = i + 1) {
     prm[i] = 1;
    }
    i = 2;           /* es prescindeix de prm[0] i de
prm[1] */
    while (i * i <= n) {
     for (j = i; j <= n / i; j = j + 1) {
         prm[j * i] = 0;
     }
     do {
         i = i + 1;
     } while (!(prm[i]));
    }
    for (i = 2; i <= n; i = i + 1) {
     if (prm[i]) {
         printf("%d\n", i);
     }
    }
    free(prm);
}
```

P2.psc

```
INICI
      ENTER m, n, r,s
      REPETIR
            LLEGEIX m, n
      FINS_QUE (m>0)&&(n>0)

      SI m<n LLAVORS
            s = Intercanvia (&m, &n)
      FI_SI

      r= m MOD n

      MENTRE r !=0 FER
            m=n; n=r; r= m MOD n
      FI_MENTRE

      ESCRIU n
FINAL

ENTER Intercanvia(ENTER *pm, ENTER *pn)
INICI_FUNCIO
      ENTER aux
      aux = *pm; *pm = *pn; *pn = aux
      RETORNA aux
FI_FUNCIO
```

P2.h

```
int Intercanvia(int *pm, int *pn);
```

P2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "P2.h"

int main(int argc, char *argv[])
{
    int m, n, r,s;
    do {
     scanf("%d", &m);
     scanf("%d", &n);
    } while (!((m > 0) && (n > 0)));
    if (m < n) {
     s = Intercanvia(&m, &n);
    }
    r = m % n;
    while (r != 0) {
     m = n;
     n = r;
     r = m % n;
    }
    printf("%d\n", n);
}
int Intercanvia(int *pm, int *pn)
{
    int aux;
    aux = *pm;
    *pm = *pn;
    *pn = aux;
    return aux;
}
```

7. CONCLUSIONS

Com a conclusió del treball realitzat es farà un breu resum del seu contingut indicant les metes aconseguides i els medis necessaris per assolir-les, així com les possibles línies futures del treball.

7.1. Fase d'estudi

En una primera fase es va realitzar l'estudi sobre les eines ja existents que podien ajudar a fer la traducció menys costosa i complexa. Així es va decidir que la millor eina per fer canvis lèxics en texts era `Lex`.

També es va tenir que estudiar possibles solucions per fer el compilador de `C`, la millor manera era utilitzar la comanda `gcc`, però per fer més fàcil la utilitat del projecte per l'usuari final es va decidir fer la trucada a `Lex` i la traducció en un `script` de `bash`.

7.2. Disseny i implementació

En aquesta fase es va començar agafant totes les paraules clau existents en el llenguatge en `pseudocodi` i traduint aquelles que eres més instantànies, deixant pel final les més complexes.

Una vegada `Lex` funcionava i traduïa les paraules menys complexes, es van anant generant totes les estructures necessàries i anant resoldre totes les necessitats que cadascuna d'elles tenia (obrir claus, tancar parèntesis, etc...).

Després es van traduir les paraules més complexes com `ESCRIU` i `LLEGEIX` que requerien saber de quin tipus eren les variables que tenien abans de traduir o `ALLOTJA` que la seva traducció depenia de quantes variables tenia definides.

I per últim es va solucionar el problema de la definició de les funcions abans de la declaració del `main` creant un text apart i incloent aquest arxiu als includes del text escrit en `C`.

Per fer l'`script` de `bash` es va pensar que la millor manera era fer que es pogués utilitzar amb les mateixes opcions que al `gcc`, d'aquest manera no seria costos entendre les utilitats i funcions d'aquest ja que el manual del `gcc` podria ajudar a futurs usuaris.

7.3. Treball Futur

Una vegada realitzada l'aplicació es suggereix com treball futur la construcció d'un analitzador sintàctic que indiqui errades escrites al text en `pseudocodi`, es podria utilitzar l'eina YACC que sol anar acompanyada de `Lex`.

8. BIBLIOGRAFIA

Documentació sobre Compiladors:

- Compilers: Principles, Techniques, and Tools
  Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman

Documentació sobre C:

- http://garota.fismat.umich.mx/mn1/manual/manual.html
- http://www2.udec.cl/~rocanale/apuntes/manual_C/index.html

Documentació sobre LEX:

- http://dinosaur.compilertools.net/lex/index.html
- http://plan9.bell-labs.com/magic/man2html/1/lex
- http://kataix.umag.cl/~mmarin/topinf/clases2003/lexCpp/index.html
- http://epaperpress.com/lexandyacc/index.html
- http://www.cs.rug.nl/~jjan/vb/lextut.pdf
- http://www.ibm.com/developerworks/library/l-lex.html?dwzone=linux

Documentació sobre Script i Bash:

- http://personales.ya.com/macprog/bash.pdf
- http://tldp.org/LDP/abs/abs-guide.pdf
- http://aplawrence.com/Unix/getopts.html
- http://www.linux-es.org/node/107
- http://club.telepolis.com/jagar1/Unix/Sed.htm
- http://linux.about.com/od/commands/l/blcmdl_sed.htm
- http://torch.cs.dal.ca/~johnston/unix/indent.html

Documentació sobre el pseudocodi:

- E.Gardeñes, J.Timoneda: Proposta de Pseudocodi. Dpt. de Matemàtica Aplicada i Anàlisi, UB.

9. ANNEXOS

9.1. Codi Lex

```
%option yylineno noyywrap

%{

//DECLARACIO D'INCLUDES
//Faran falta per poder utilitzar les funcions que fem servir en el codi
//d'aquest arxiu

#include <stdio.h>
#include <stdlib.h>
#include <string.h>


//DECLARACIONS DE VARIABLES

//Guarda el valor de yytext analitzat anteriorment
char yytextAnterior [100];
//0:no hem de tancar parèntesis, >0: hem de tancar parèntesis
int tancarParentesi;
//0:no hem d'obrir clau, >0: hem d'obrir clau
int obrirClau;
//0:no hem de tancar clau, >0: hem de tancar clau
int tancarClau;
//0:no estem en scanf o printf, 1: estem en scanf 2: estem en printf
int escriuTipoVar;
//0:no s'ha imprès scanf o printf,
//>0: indica el numero de vegades que s'ha imprès scanf o printf
int numImpTipoVar;
//0:no s'ha d'imprimir yytext, 1: si s'ha d'imprimir yytext
int impyytext;
//0:no s'ha de multiplicar malloc (fa ALLIBERA amb 1 paràmetre)
//1: si s'ha de multiplicar malloc (fa ALLIBERA amb 2 paràmetres)
int multMalloc;
//>0: indica el numero de case que s'han escrit
int intCase;

//Indica la posició del nou element a l'arrayint
int yint;
//Indica la posició del nou element a l'arraychar
int ychar;
```

```
//Indica la posició del nou element a l'arraydouble
int ydouble;
//Indica la posició del nou element de l'array frase
int yfrase;
//Indica la posició del nou element de l'array fraseAnterior
int yfraseAnterior;

//Guarda la variable que s'ha declarat,pot ser: int,char o double
char tipoVar [10];
//Indica la variable del for
char varfor[50];
//Indica la variable del malloc
char varmalloc[50];

//1: Indica que el document comença amb INICI
int intInici;
//1: Indica que hem arribat a la paraula FINAL
int intFinal;
//1: Indica que estem dins d'una funció
int intFuncio;
//1: Indica que hem imprès un operador
int intImpOperador;

//Arxiu per guardar Funcions.h
FILE *arxiuFuncions;

// Variable pels arrays d'strings
typedef char var[50];

//Arrays per guardar els diferents tipus de variables
var arrayint[100]={};
var arraychar[100]={};
var arraydouble[100]={};

//Arrays per guardar les frases completes
//(per poder imprimir les declaracions de les funcions)
var frase[100];                 //Guarda la frase que escriurem
var fraseAnterior[100];   //Guarda la frase que he escrit anteriorment


//DECLARACIONS DE FUNCIONS

//Si existeix un error imprimeix per pantalla quin és i la línia
//on es troba
void yyerror(char *s);
```

```
//Crear l'arxiu Funcions.h
void CrearArxiuFuncions();

//Imprimeix l'operador depenent si estem en un fins que o no
void ImprimirOperador(int);

//Fa un printf de la Cadena i reseteja la l'array frase si cal
void ImprimirParaula(char Cadena[100]);

//Posa en blanc l'array frase  i omple l'array fraseAnterior
void ReseteaFrase();

//Indica l'ultima paraula que estem analitzant
void UltimaParaula(char Cadena[100]);

//Escriu la declaracio de la funció a l'arxiu Funciones.h
void GuardaFuncions();

//Guarda la variable que s'està declarant a l'array corresponent
void GuardaVariables();

//Imprimeix la variable quan hem fet un LLEGEUIX o ESCRIU
//posant el tipus que es
void ImprimirTipoVar();

//S'utilitza quan terminem de llegir una línia, comprova
//si falta tancar parèntesis, claus, obrir-los, posar ; etc..
void finalLinea();


%}
%%



[a-z]        {UltimaParaula(yytext);          //Qualsevol lletra
              ImprimirParaula(yytext);}

[0-9]+       {UltimaParaula(yytext);          //Qualsevol numero
              ImprimirParaula(yytext);} //de 1 o mes longitud

"ENTER"      {ImprimirParaula("int");         //Declaracions
              UltimaParaula(yytext);
              strcpy(tipoVar,"int");}
"REAL"       {ImprimirParaula("double");      //Declaracions
              UltimaParaula(yytext);
```

```
              strcpy(tipoVar,"double");}
"CAR"         {ImprimirParaula("char");        //Declaracions
              UltimaParaula(yytext);
              strcpy(tipoVar,"char");}
"CONST"             {ImprimirParaula("const");      //Constants
              UltimaParaula(yytext);}

"LLEGEIX"     {ImprimirParaula("scanf(");    //Instruccions elementals
              escriuTipoVar=1;         //Scanf=1, printf=2
              numImpTipoVar=0;     //Encara no s'ha imprès
              tancarParentesi++;   //Necessita tancar parèntesi
              UltimaParaula(yytext);}

"ESCRIU"      {ImprimirParaula("printf(");   //Instruccions elementals
              escriuTipoVar=2;   //Scanf=1, printf=2
              numImpTipoVar=0;   //Encara no s'ha imprès
              tancarParentesi++; //Necessita tancar parèntesi
              UltimaParaula(yytext);}

"SI"          {ImprimirParaula("if (");        //Bloc condicional
              UltimaParaula("(");}

"LLAVORS"     {ImprimirParaula(") {");  //Bloc condicional
              UltimaParaula("{");
              tancarClau++;}

"SI_NO"       {                         //Bloc condicional
              //Si s'ha se tancar clau
              if (tancarClau>0) ImprimirParaula("}");
              ImprimirParaula("else");
              //Si s'ha tancat hem d'obrir clau
              if (tancarClau>0) ImprimirParaula("{");
              UltimaParaula("else");}

"FI_SI"       {ImprimirParaula("}");    //Bloc condicional
              UltimaParaula("}");
              //Al finalitzar la línia tancarem clau
              tancarClau--;}

"SEGONS"      {ImprimirParaula("switch (");    //Alternativa Múltiple
              tancarParentesi++;
              obrirClau++;
              UltimaParaula("switch");}
```

```
"CAS"         {intCase++;
               if (intCase>1) {               //El primer no
                     ImprimirParaula("break;");
                     intCase--;
               }
               UltimaParaula("case");         //Alternativa Múltiple
               ImprimirParaula("case");
               }

"ALTRAMENT"   {if (intCase>0) {         //Alternativa Múltiple
                     ImprimirParaula("break;");
                     intCase--;
               }
               UltimaParaula("default");//Alternativa Múltiple
               ImprimirParaula("default");
               }

"FI_SEGONS"   {if (intCase>0) {         //Alternativa Múltiple
                     ImprimirParaula("break;");
                     intCase--;
               }
               UltimaParaula("}");
               ImprimirParaula("}");    }

"MENTRE"      {//Iteració condicional final
               ImprimirParaula("while (");
               UltimaParaula("(");}

"FER"         {//Iteració condicional final
               ImprimirParaula("){");
               UltimaParaula("{");}

"FI_MENTRE"   {//Iteració condicional final
               ImprimirParaula("}");
               UltimaParaula("}");}

"REPETIR"     {//Iteració condicional final
               ImprimirParaula("do {");
               UltimaParaula("{");}

"FINS_QUE"    {//Iteració condicional final
               ImprimirParaula("} while (");
               //Neguem, són contaries
               ImprimirParaula("!(");
               //Necessita tancar 2 parèntesis
               tancarParentesi=tancarParentesi+2;
```

```
              UltimaParaula("(");}

"PER"|"DES_DE"      {//Iteració indexada
              ImprimirParaula("for (");
              //Necessita tancar paréntesis
              tancarParentesi++;
              //Necessita obrir clau
              obrirClau++;
              UltimaParaula("for (");}

"FINS_A"     {//Iteració indexada
              ImprimirParaula(";");
              UltimaParaula("FINS_A");}

"INCR"|"INCREMENT"|"CADA" {//Iteració indexada
                           ImprimirParaula(";");
                           UltimaParaula("INCR");}


"FI_PER"|"FI_DES_DE"      {//Iteració indexada
                     ImprimirParaula("}");
                     UltimaParaula("}");}

"ESTRUCTURA" {ImprimirParaula("struct");      //Estructures
              obrirClau++;             //Necessita obrir clau
              UltimaParaula(yytext);}

"FI_ESTRUCTURA"     {ImprimirParaula("}");           //Estructures
              //Tanca la clau que hem obert a
              UltimaParaula(yytext);}

"ALLOTJA"|"ALLOTJAR"|"RESERVA"|"RESERVAR"     {//Assignació Dinàmica
                                        //de  Memòria
                                       ImprimirParaula("");
                                       UltimaParaula("ALLOTJA");
                                       }
"ALLIBERA"|"ALLIBERAR"           {//Assignació Dinàmica de  Memòria
                           ImprimirParaula("free (");
                           //Necessita tancar paréntesis
                           UltimaParaula("(");}


"GLOBAL"     {ImprimirParaula("");            //Seccions
              UltimaParaula("\n");}

"FI_GLOBAL"  {ImprimirParaula("");     //Seccions
              UltimaParaula("\n");}
```

```
"INICI"            {intInici=1;       //Hem passat per un INICI
             ImprimirParaula("#include <stdio.h> \n#include
             ImprimirParaula("<stdlib.h> \n");
             ImprimirParaula("#include     \"Funciones.h\" \n\n");
             ImprimirParaula("int main (int argc, char *argv[]) {");
             UltimaParaula("}");}

"FINAL"            {ImprimirParaula("}");          //Seccions
              intFinal=1;             //Hem passat per un FINAL
              UltimaParaula("}");}

"INICI_FUNCIO"     {intFuncio=1;             //Seccions
              ImprimirParaula("{");
              GuardaFuncions();
              UltimaParaula("{");}

"FI_FUNCIO"  {intFuncio=0;             //Seccions
              ImprimirParaula("}");
              UltimaParaula("}");}


"SURT"|"SORTIR"    {ImprimirParaula("return;");     //Terminació
              UltimaParaula(";");}


"ABANDONA"   {ImprimirParaula("break;");      //Terminació
              UltimaParaula(";");}


"CONTINUA"   {ImprimirParaula("continue;");   //Terminació
              UltimaParaula(";");}

"RETORNA"    {ImprimirParaula("return");      //Terminació
              UltimaParaula("return");}


"DIV"        {ImprimirParaula("/");    //Operador
              UltimaParaula("/");}

"MOD"        {ImprimirParaula("%");    //Operador
              UltimaParaula("%");}

"O"          {UltimaParaula("||");     //Operador
              ImprimirParaula("||");}

"I"          {ImprimirParaula("&&");   //Operador
```

```
              UltimaParaula("&&");}

"NO"          {ImprimirParaula("!");    //Operador
              UltimaParaula("!");}

"/*"          {finalLinea();}     //Comentari: mirem si cal variables

"*/"          {UltimaParaula(" ");
              //Final comentari: no posar punt i coma
              ImprimirParaula("*/");}


"}"|"{"       {UltimaParaula(yytext);}  //Claus

"("|")"       {UltimaParaula(yytext);          //Parèntesi
              ImprimirParaula(yytext);}

"!|"          {UltimaParaula(yytext);          //Exclamacions
              ImprimirParaula(yytext);}

""|"?"        {UltimaParaula(yytext);          //Interrogacions
              ImprimirParaula(yytext);}

"+"|"-"|"/"|"*"|"=" {UltimaParaula(yytext);          //Operadors
                    ImprimirParaula(yytext);}

"&"|"$"|"."|":"|"_"|"<"|">"      {UltimaParaula(yytext);          //Varis
                          ImprimirParaula(yytext);}

","           {//Guarda les variables si estem en 1 declaració
              GuardaVariables();
              //Si ve d'un malloc es que te
              //+ d'una variable
              if (strcmp(varmalloc,"")!=0) multMalloc=1;
                    // Si no estem en un printf o scanf i existeix
                    //una variable malloc ja definida
                    if ((escriuTipoVar==0)
                          &&(strcmp(varmalloc,"")==0)) {
                          ImprimirParaula(",");
                    }
              }

";"           {//Punt i coma (te un comportament diferent)
              ImprimirParaula(";\n");
              UltimaParaula(yytext);}
```

```
" "          {ImprimirParaula(yytext);}

[a-zA-Z][a-zA-Z0-9]+      {//Qualsevol paraula
                        UltimaParaula(yytext);
                        ImprimirParaula(yytext);}


\t           {ImprimirParaula("");
              UltimaParaula(yytext);}


\n           {//Per indicar el numero de línia
              yylineno = yylineno + 1;
              finalLinea();}

%%

//Main que s'encarrega de cridar a la funció interna del lex: yylex()
main() {

       //Creem l'arxiu "Funcions.h"
       CrearArxiuFuncions();

       //Inicialització de Variables
       yylineno=1;
       tancarParentesi=0;
       obrirClau=0;
       tancarClau=0;
       escriuTipoVar=0;
       yint=0;
       ychar=0;
       ydouble=0;
       numImpTipoVar=0;
       impyytext=1;
       multMalloc=0;
       yfrase=0;
       intInici=0;
       intFinal=0;
       intFuncio=0;
       intImpOperador=0;
       intCase=0;

       //Crida a la funció principal del Lex
       return yylex();
}
```

```
//Funció que imprimeix per pantalla un error i la línia on es troba
void yyerror (char *msg){
	fprintf(stderr,"Linea %d: %s en '%s'\n", yylineno, msg, yytext);
}

//Funció que crea l'arxiu de Funcions.h per poder escriure en ell
//quan trobem una declaració
void CrearArxiuFuncions()
{
	arxiuFuncions=fopen("Funcions.h","w");
	if (!arxiuFuncions){
		printf("Error a l'obrir el fitxer de funcions\n");
		return;
	}
}

//Funció que imprimeix una paraula al text de sortida
void ImprimirParaula(char Cadena[100])
{
	if (impyytext==1) {
		printf("%s",Cadena);
		strcpy(frase[yfrase],Cadena);
		yfrase++;
	}

	//Si és final de línia comencem una nova frase
	if ((strcmp(Cadena,"\n")==0) || (strcmp(Cadena,";\n")==0)){
		ReseteaFrase();
	}
}

//Funció que posa en blanc l'array frase i omple l'array fraseAnterior
void ReseteaFrase(){

	//Hem acabat la frase, la posem en blanc per guardar una nova
	int x;
	for (x=0; x<yfrase;x++){
		strcpy(fraseAnterior[x],frase[x]);
		strcpy(frase[x],"");
	}

	yfraseAnterior=yfrase;
	yfrase=0;
}
```

```
//Funció que escriu la declaració de la funció a l'arxiu Funciones.h
void GuardaFuncions()
{
	int x;

	for (x=0; x<yfraseAnterior;x++){
		//Copia la declaració de la funció a "Funcions.h"
		fprintf(arxiuFuncions,fraseAnterior[x]);
	}
	fprintf(arxiuFuncions,";\n");

}

//Funció que indica l'ultima paraula que estem analitzen
//Avanç de donar-li valor a la actual mirem si l'anterior necessita
//alguna condició especial
void UltimaParaula(char Cadena[100])
{
	//Per defecte sempre imprimirem el text
	impyytext=1;

	//Si l'anterior paraula és for, jo soc la variable
	if (strcmp(yytextAnterior,"for (")==0){
		strcpy(varfor,Cadena);
	}

	//Si és INCR de FOR hem de posar el tipus var
	if (strcmp(yytextAnterior,"INCR")==0){
		printf("%s=%s",varfor,varfor);
	}

	//Si és FINS_A de FOR hem de posar el tipus var <=
	if (strcmp(yytextAnterior,"FINS_A")==0){
		printf("%s<=",varfor);
	}

	//Si és ALLOTJA hem de posar = malloc(
	if (strcmp(yytextAnterior,"ALLOTJA")==0){
		printf("%s = malloc((",Cadena);
		//No te que imprimir la cadena,ja la hem imprès
		impyytext=0;
		//Copiem la paraula declarada
		strcpy(varmalloc,Cadena);
	}
```

```
	//Si hem arribat al final del fitxer i encara no hem declarat
	//la funció i és la primera paraula de la frase mirem si és
	// char, int o double,
	//si no és aixi necessita posar void
	if ((intFinal==1) && (intFuncio==0)
		&& (strcmp(yytextAnterior," ")==0)) {
		if ((strcmp(Cadena,"ENTER")!=0)
			&& (strcmp(Cadena,"REAL")!=0)
			&& (strcmp(Cadena,"CAR")!=0)
			&& (strcmp(Cadena,"}")!=0)
			&& (strcmp(Cadena," ")!=0)) {

			ImprimirParaula("void ");

		}
	}

	//Copiar la variable anterior
	strcpy(yytextAnterior,Cadena);

	//Mirem si hem de fer printf o scanf amb una variable
	ImprimirTipoVar();
}

//Funció que guarda les variables en el seu determinat array
//si existeix declaració
void GuardaVariables(){

	if (strcmp(tipoVar,"int")==0){
		strcpy(arrayint[yint],yytextAnterior);
		yint++;
	}else if (strcmp(tipoVar,"char")==0) {
		strcpy(arraychar[ychar],yytextAnterior);
		ychar++;
	}else if (strcmp(tipoVar,"double")==0) {
		strcpy(arraydouble[ydouble],yytextAnterior);
		ydouble++;
	}


}

//Funció que imprimeix la variable quan hem fet un LLEGUEIX o
//ESCRIU posant el tipus que es
void ImprimirTipoVar()
{
```

```
int i;

//Mirem si estem fent un scanf o printf
if (escriuTipoVar>0){
	if ((strcmp(yytextAnterior,"LLEGEIX")!=0)
		&& (strcmp(yytextAnterior,"ESCRIU")!=0)
			&& (strcmp(yytextAnterior,")")!=0)
			 && (strcmp(yytextAnterior," ")!=0)){

		if (numImpTipoVar>0){
			//Si ja hem imprés una variable tornem
			//a posar printf
			printf(");\n");
			//Imprimim scanf o printf
			//una altre vegada
			if (escriuTipoVar==1){
				ImprimirParaula("scanf(");
			}else{
				ImprimirParaula("printf(");
			}
			//Augmentem les variables impreses
			numImpTipoVar++;
		}else{
			//Ara imprimirem una
			numImpTipoVar=1;
		}
		ImprimirParaula("\"%");

		//Escollir el tipus de variable que es
		for (i=0; i<=yint; i++){
		    if (strcmp(arrayint[i],yytextAnterior)==0){
				ImprimirParaula("d");
		    }
		}
		for (i=0; i<= ychar ;i++){
		    if (strcmp(arraychar[i],yytextAnterior)==0){
				ImprimirParaula("s");
		    }
		}
		for (i=0; i<= ydouble ;i++){
		   if (strcmp(arraydouble[i],yytextAnterior)==0){
				ImprimirParaula("f");
		   }
		}
```

```
                        //Depenen si és un printf o scanf acabaran
                        //d'una manera
                        if (escriuTipoVar==1){
                              ImprimirParaula("\",&");
                        }else{
                              ImprimirParaula("\\n\",");
                        }

                  }
            }

}


//Funció que mira si al finalitzar una línia és necessari tancar algun
//parèntesi o clau o posar ;
//Resetea les variables que indiquen propietats de la línia a escriure
void finalLinea()
{

      //Arribats a aquest punt sempre imprimirem la paraula
      impyytext=1;

      //Mirem si la segent paraula ve d'un ALLOTJA
      if  (strcmp(varmalloc,"")!=0){
            //Si hi ha mes d'una variable es tanca ja
            //la clau i escrivim sizeof
            if (multMalloc==1) {
                  ImprimirParaula(")*");
            }
            printf(" sizeof (*%s))",varmalloc);
            //Si només hi ha 1 variable la clau es tanca
            //després d'escriure sizeof
            if (multMalloc==0) {
                  ImprimirParaula(")");
            }
      }

      //Comprovem si fa falta tancar parèntesi
      int i;
      while(tancarParentesi>0)
      {
            //Imprimeix el parèntesi
            ImprimirParaula(")");
            UltimaParaula(")");
            tancarParentesi--;
```

```
}

//Comprovem si fa falta obrir clau
if (obrirClau>0){
	ImprimirParaula("{");
	UltimaParaula("{");
	obrirClau--; //Ja hem imprès una clau
}else{
	//Comprovem si fa falta ;
	//Els casos que no necessiten son: },{,;,else i res
	if ((strcmp(yytextAnterior,"}")!=0)
		&& (strcmp(yytextAnterior,"{")!=0)
		&& (strcmp(yytextAnterior,";")!=0)
		&& (strcmp(yytextAnterior," ")!=0)
		&& (strcmp(yytextAnterior,"else")!=0)){
		GuardaVariables();

		//Si estem declarant una funció va sense ;
		//Opcions:
		//1.- No hem declarat inici: tot són funcions
		//2.- Existeix inici i final
		//Y no hem començat encara a escriure la funció
		if (((intInici==1) && (intFinal==1)
			&& (intFuncio==0)) //2
			|| ((intInici==0)
			&& (intFuncio==0))){	//1
			ImprimirParaula("");
		}else{
			ImprimirParaula(";");
		}
	}

}
//Final de línia: Si hi ha un comentari no hi haura enter
if (strcmp(yytextAnterior,"*/")==0){
	ImprimirParaula("\n");
}

//Si venim de un printf o scanf posem \n
if ((numImpTipoVar>1) && (escriuTipoVar==2)){
	ImprimirParaula("\nprintf(");
	ImprimirParaula("\"\\n\");");
}

//Posar a 0 las variables de línia
UltimaParaula(" ");
```

```
        strcpy(tipoVar,"");
        strcpy(varfor,"");
        strcpy(varmalloc,"");
        escriuTipoVar=0;
        numImpTipoVar=0;
        impyytext=1;
        multMalloc=0;
        intImpOperador=0;

        //Comencem una nova línia
        ReseteaFrase();


        ECHO;
}
```

9.2. Codi Script

```
#!/bin/bash
#Script que tradueix un arxiu escrit en pseudocodi y el compila en C per
#crear l'executable
#La forma de compilar l'arxiu es pot escollir posant diferents opcions


#Agafem les opcions passades per paràmetre per veure si ens han indicat
#la forma de compilar
opcio_O="0"
opcio="1"

#Mirem les opcions que ens passen per paràmetres
while getopts ":c:t:o" opt
do
     case $opt in

       'c')
             opcio="C";;
       't')
             opcio="T";;
       'o')
             opcio_O="1";;

     esac
done

#Recorrem tots els paràmetres y traduïm a .c tots el arxius .psc
for i in $*; do
       #Si no són opcions de compilar
       if [ $i != '-c' ] && [ $i != '-o' ] && [ $i != '-t' ]; then

             #Primer hem de treure l'extensió que ens passen per
             #agafar el nom del arxiu
             arxiu=$i
             nomArxiu=${arxiu%.*}

             #Només agafem els psc y fem la traducció
             case $i in
                  *.psc)
                     #Passem l'executable del traductor que crea
                     #l'arxiu .c y Funciones.h
                     ./Traductor < $i > $nomArxiu_1.c

                     #Canviem el nom Funcions.h pel nom de l'arxiu
                     #passat per paràmetre
                     mv Funcions.h $nomArxiu.h

                     #Canviem el texte Funcions.h en l'arxiu C pel nou nom
                     sed s/Funciones.h/$nomArxiu.h/g $nomArxiu_1.c >
                     $nomArxiu.c




                     #Arreglem el codi perquè quedi mes tabulat
                     indent -kr $nomArxiu.c

                     #Creem el .o i el guardem a la variable de .o
                     gcc -c $nomArxiu.c;
                     OBJ=$OBJ" "$nomArxiu.o

                     #Mirem si l'arxiu .h no és vuit, si ho és hem
                     #d'eliminar-lo
                     if ! [ -s $nomArxiu.h ]; then
```

```
                        #Eliminem l'arxiu
                        rm $nomArxiu.h

                        #Trèiem l'include que esta sempre a la línia 3
                        mv $nomArxiu.c $nomArxiu_1.c
                        sed -e '3d' $nomArxiu_1.c > $nomArxiu.c
                    fi

                    #Esborrem l'arxiu _1.c que es va crear perquè en el
                    #mateix #arxiu no podíem canviar Funcions.h
                    rm $nomArxiu_1.c;;
                *.c)
                    #Crem el .o i el guardem a la variable de .o
                    gcc -c $nomArxiu.c;
                    OBJ=$OBJ" "$nomArxiu.o;;
                *.o)
                     #Guardem a la variable de .o
                     OBJ=$OBJ" "$nomArxiu.o;;
                *)
                     #Serà el nom de l'executable
                     arxiuExe=$nomArxiu;;
            esac
        fi
done

#Mirem si hem de compilar

if [ $opcio == 'T' ]; then
        #Esborrem els .o perquè no tenien que haver-se creat
        rm $OBJ
else
        if [ $opcio != 'C' ]; then
            #Si no és ni -c ni -t tenim que compilar i linkar els objects
            #Si han posat la opció -o l'arxiu Exe serà el de per defecte
            if [ "$arxiuExe" = "" ]; then
                  arxiuExe="a.out"
            fi
            #Compilem
            gcc -o $arxiuExe $OBJ;
        fi
fi
```

ENGINYERIA TÈCNICA EN INFORMÀTICA DE SISTEMES

UNIVERSITAT DE BARCELONA

Treball fi de carrera presentat el dia de de 200
a la Facultat de Matemàtiques de la Universitat de Barcelona,
amb el següent tribunal:

Dr. President

Dr. Vocal

Dr. Secretari

Amb la qualificació de: