Presentació

En aquesta Prova d'Avaluació Continuada es treballa l'estructura general d'un compilador i, més concretament l'etapa d'anàlisi lèxic. La PAC combina preguntes teòriques amb un exercici pràctic on haureu de fer servir el software JLex per construir un analitzador lèxic.

Competències

En aquesta PAC es desenvolupen les següents competències del Grau en Enginyeria Informàtica:

- Capacitat per analitzar un problema en el nivell d'abstracció adequat a cada situació i aplicar les habilitats i coneixements adquirits per a resoldre'l.
- Capacitat per dissenyar i construir aplicacions informàtiques mitjançant tècniques de desenvolupament, integració i reutilització.
- Capacitat per aplicar les tècniques específiques de tractament, emmagatzemament i administració de dades.
- Capacitat per proposar i avaluar diferents alternatives tecnològiques per resoldre un problema concret.

Objectius

Els objectius concrets d'aquesta Prova d'Avaluació Continuada són:

- Tenir una visió històrica dels llenguatges de programació.
- Conèixer l'estructura bàsica d'un compilador.
- Saber utilitzar els diagrames de Tombstone per descriure el procés de construcció d'un compilador.
- Entendre el paper de l'analitzador lèxic en un compilador.
- Repassar conceptes bàsics d'expressions regulars com a mecanisme per descriure un llenguatge.

- Conèixer com s'implementa un analitzador lèxic en un compilador.
- Conèixer els objectius d'una taula de símbols i les diferents estratègies per implementar-les.
- Aprendre a descriure analitzadors lèxics utilitzant JLex.

Descripció de la PAC

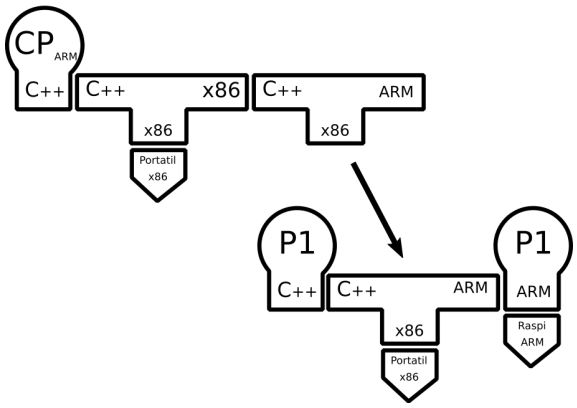
Exercici 1 (10%)

La *cross-compilation*, és un procés a partir del qual utilitzant un compilador que funciona sobre una arquitectura concreta es poden generar binaris per una arquitectura diferent. Suposem que volem compilar el nostre programa C++ anomenat **P1** utilitzant el nostre ordinador portàtil, basat en l'arquitectura **x86**, per a ser utilitzat en una *Raspberry Pi*: un mini-ordinador de baixes prestacions basat en arquitectura **ARM**. Per a fer-ho disposem dels següents elements:

- El codi font del programa P1, escrit en C++
- El **codi font** d'un compilador de C++ que emet codi màquina ARM
- Un compilador de C++, que emet codi màquina x86 i s'executa sobre la mateixa arquitectura
- Un ordinador portàtil basat en arquitectura x86
- Una *Raspberry pi*, basada en arquitectura ARM

Dibuixa els diagrames de Tombstone que, utilitzant els elements anteriorment descrits, permeten l'execució de P1 sobre la *Raspberry Pi*. Descriu breument el significat de cadascun dels símbols utilitzats en el diagrama.

Solució:



Exercici 2 (30%)

Respon les següents preguntes.

1. Indica, per a cadascuna de les següents funcionalitats de llenguatges de programació, quina de les fases del *front-end* d'un compilador té més pes: la lèxica, la sintàctica o la semàntica. S'han de justificar les respostes:

   - La presència de *first-class functions*[1]
   - Els comentaris multilínia utilitzant `/* */`
   - L'ús de parèntesi per agrupar expressions aritmètiques
   - Distinció entre majúscules i minúscules als identificadors
   - Permetre la definició de classes i interfícies
   - Utilitzar punt i coma `;` com a separador de línia

[1] La capacitat de tractar funcions com objectes i passar-los com a paràmetre d'altres funcions. Més informació: `https://en.wikipedia.org/wiki/First-class_function`

2. Indica la diferència entre un compilador i un intèrpret de codi. La màquina virtual de Java[2] és un compilador o un intèrpret? Justifica la teva resposta.

3. Escriu expressions regulars per a reconèixer les següents estructures lèxiques. Per a fer-ho, utilitza la sintaxi de definició d'expressions regulars de JLex:

   (a) Reconèixer números de telèfon. Un número de telèfon consisteix en un seguit de 9 dígits. Aquests dígits poden contenir opcionalment guions o espais entre cada parella de dígits, però en cada nombre només hi pot haver un dels dos estils: o bé guions, o bé espais. A més, opcionalment, un número de telèfon pot anar precedit d'un codi de país, que s'indica amb el símbol `+`, seguit de dos o tres dígits. Exemples: `555-555-555`, `902 123 456`, `+34 45 677 89 10`, `9-8-7-6-5-4-3-2-1`, `123456789`.

   (b) Contrassenyes segures. Una contrassenya segura és un seguit de caracters alfanumèrics (lletres de l'alfabet i dígits del 0 al 9) i caràcters especials, d'entre els quals considerem: `!`, `%`, `#` i `/`. A més, s'estableix el requisit que una contrassenya segura conté almenys un dígit i un caràcter especial. Exemples: `contrassenyasegura!123`, `FLD9t/Ct8PHPq%gJ`

Solució:

Pregunta 1

1. La presència de *first-class functions* és una funcionalitat semàntica, ja que és un concepte que es pot implementar en un llenguatge independentment de la sintàxi que se li doni. A més, el gruix principal de l'implementació recau a l'analitzador semàntic, ja que creix la complexitat del sistema de tipus, i cal analitzar el fenòmen de la captura de variables en clausures.
2. Els comentaris són un constructe lèxic. S'eliminen de l'anàlisi a l'inici del tot, juntament amb l'espai en blanc no significatiu i no tenen cap pes a les fases semàntica i sintàctica.
3. L'ús de parèntesi és un constructe sintàctic. Seria un error tractar-lo a la part lèxica, ja que per detectar aquest tipus d'expressions cal una gramàtica lliure de context. Generalment els parèntesis només serveixen per crear l'arbre de parsing amb la precedència correcta, i s'ignoren a la fase d'anàlisi semàntica.
4. La distinció entre majúscules i minúscules és tracta a la fase lèxica del llenguatge, on caldrà decidir si a les expressions regulars es tracten les majúscules i minúscules com a caràcters diferents.

[2] Més informació: `https://en.wikipedia.org/wiki/Java_virtual_machine`

5. La definició de classes i interfícies no depèn del lèxic i sintaxi que tinguin associats. De manera similar al que passava amb les `first-class function`, el pes d'aquest anàlisi recau a la part semàntica, ja que cal introduïr nous camps a la taula de símbols i s'afegeix complexitat al sistema de tipus.
6. L'ús de punt i coma per dividir línies és una característica sintàctica, ja que restringeix l'ordre en que el compilador espera veure els diferents tokens. A nivell lèxic l'impacte és molt petit: només cal afegir el punt i coma com a nou token.

Pregunta 2

Un compilador és un programa que transforma el codi font, escrit en un llenguatge de programació d'alt nivell, en codi màquina: binari pensat per ser executat directament per un processador. En canvi, un intèrpret és un programa que executa el codi font directament, sense transformar-lo a codi màquina.

La màquina virtual de Java (JVM), és el que es coneix com un intèrpret de *bytecode*. És un sistema en dues parts: D'una banda es defineix una arquitectura hardware fictícia, amb el seu joc d'instruccions i especificacions. El codi màquina d'aquest sistema és el que s'anomena *bytecode*, i existeix un compilador, `javac`, que transforma fitxers de codi font Java a aquest *bytecode*. D'altra s'implementa un software que funciona com un emulador d'aquesta arquitectura ficticia, la màquina virtual.

Així doncs, direm que Java és un llenguatge compilat "a *bytecode*" i que la JVM és un intèrpret d'aquest llenguatge binari de baix nivell, però no de Java.

Pregunta 3

1.`(\+([0-9][0-9]|[0-9][0-9][0-9])) (([0-9] ?[0-9] ?[0-9] ?[0-9] ?[0-9] ?[0-9] ?[0-9] ?[0-9] ?[0-9]|[0-9]-?[0-9]-?[0-9]-?[0-9]-?[0-9]-?[0-9]-?[0-9]-?[0-9]-?[0-9])`

En l'expressió regular, distingim dues parts. En primer lloc el prefix `+([0-9][0-9]|[0-9][0-9][0-9]))` que indica el codi de països. Després tenim una alternativa entre el nombre de telèfon, amb separadors opcionals, ja sigui utilitzant espais o guions, però no barrejant-los.

Alternativament, es podria simplificar utilitzant la versió abreviada `{N}` que indica un nombre concret de repeticions, i la construcció `\d` que substitueix `[0-9]`:

1.1. `(\+(\d{2}|\d{3}) )(((\d ?){8}\d|(\d-?){8}\d)`

2. `([a-zA-Z0-9!%#/]*[0-9][a-zA-Z0-9!%#/]*[!%#/][a-zA-Z0-9!%#/]* |`

   `[a-zA-Z0-9!%#/]*[!%#/][a-zA-Z0-9!%#/]*[0-9][a-zA-Z0-9!%#/]*)`

Aquesta expressió regular es defineix amb dos casos. En tots dos utilitzem `[a-zA-Z0-9!%#/]` per indicar

qualsevol dels caràcters de la contrassenya. `(i)` En el primer cas, s'indica que, entremig d'una cadena arbitràriament llarga de caràcters vàlids, apareixerà primer un dígit, i després un símbol especial. `(i)` En el segon cas, es fa al revés: Primer es mira si apareix el caràcter especial, i després el dígit. Amb aquests dos casos es cobreixen totes les possibilitats.

Exercici 3 (60%)

Durant el curs, treballarem en el desenvolupament del compilador d'un llenguatge per a la definició de serveis per una aplicació de domòtica. En aquesta primera PAC, ens centrarem en el desenvolupament de l'analitzador lèxic del llenguatge.

A nivell lèxic, el llenguatge està format pels següents elements:

Paraules Reservades: `audio`, `switch`, `json`, `script`, `service`, `plays`, `opens`, `closes`, `int`, `string`, `endpoint`, `request`, `say`, `let`, `sleep`

Operadors: `+`, `-`, `*`, `/`, `=`, `++`,

Delimitadors: {, }, `(`, `)`, `;`, `:`, `,`

Comentaris: Els comentaris s'indiquen amb el caràcter `#`. Tot el que hi ha des d'un delimitador de comentari fins al proper salt de línia és part del comentari. Exemple:
`# This is a simple comment`

Identificadors: Els identificadors vàlids consisteixen en una seqüència de caràcters alfanumèrics, i barra baixa (_). A més, s'afegeix la restricció que un identificador mai pot començar per un dígit. Exemples: `product_id`, `PlayAlarmSound`, `__some_variable`

Nombres Enters: Es tracta únicament d'enters (sense part decimal). Opcionalment precedits pel signe `-` per indicar negatiu. Exemples: `25`, `-3`, `0`.

String: Cadenes de text arbitrari, delimitades per cometes dobles `"`. **No** hi ha sequències d'escapament dins les string, com `\n` o `\"`. Això implica que les string tampoc poden contenir el seu delimitador. Exemple: `"this is a string"`.

Rutes de fitxer: Les rutes de fitxer indiquen la posició d'un arxiu dins un sistema de fitxers, s'utilitza la convenció d'estil UNIX per especificar-los:

- Una ruta de fitxer consisteix en un o més *noms de directoris*, separats per *barres* `/`

- Opcionalment, es pot afegir una barra addicional a l'inici per indicar rutes absolutes
- Obligatòriament, després de l'últim separador de directori, s'ha d'incloure un nom de fitxer.
- Els noms de fitxers i directoris poden consistir de caràcters alfanumèrics, barres baixes (_), dígits i el caràcter punt (.) Fixeu-vos que, a diferència dels identificadors, aquí està permès que el primer caràcter d'un nom de fitxer sigui un dígit
- Davant l'ambigüitat entre una expressió aritmètica de divisió i una ruta de fitxer, la interpretació de ruta de fitxer ha de tenir preferència. Per indicar divisió s'haurà d'especificar amb un espai entre el signe de divisió i els operands.

Exemples de rutes de fitxer: `/home/josep/UOC/PAC1.pdf`, `/usr/lib64/libfreetype.so`

Adreces Web: Les adreces web indiquen la direcció d'un recurs al web. Son similars a les rutes de fitxer, amb algunes particularitats:

- Una direcció web comença amb una cadena de `protocol`, que podrà ser `http` o bé `https`, seguida de dos punts `:` i dues barres `//`.
- A continuació, segueix un `domini web`, que consisteix en un seguit d'un o més *noms*, separats per punt `.`. Els noms dins el domini web consisteixen en un seguit de caràcters alfanumèrics, amb dígits, majúscules i minúscules. En aquest cas, també es permet que la cadena comenci per un dígit.
- L'últim segment del domini, és un `top-level domain`. Per simplificar, direm que els top level domain són els següents: `com`, `org`, `net`, `cat`, `es`, `de`, `fr`, `it`.
- Seguit del domini, opcionalment, pot aparèixer una barra (`/`), seguida d'una `ruta dins el domini`. Les rutes d'un domini consisteixen en un seguit de noms, tal com s'han definit a l'apartat de *domini web*, separats entre sí per barres (`/`), i podent acabar opcionalment amb un caràcter `/`.

Exemples de direccions web: `https://www.uoc.edu`,
`http://subdomini.elmeudomini.cat/projectes/projecte25`

El llenguatge ha de ser **sensible a les majúscules**. Així, per exemple, l'entrada `SERVICE` es llegirà com un identificador, mentre que `service` serà una paraula reservada.

Es demana que, mitjançant JLex, desenvolupeu un analitzador capaç de reconèixer les estructures lèxiques del llenguatge. En la mesura que sigui possible, l'analitzador ha de detectar i recuperar-se d'errors lèxics. A més, l'analitzador haurà de comptar i escriure a la sortida el **nombre de paraules reservades** i el **nombre d'identificadors detectats**, així com el **nombre d'errors lèxics**.

Per aquesta fase d'anàlisi s'han d'utilitzar expressions regulars mitjançant JLex. Es valorarà negativament l'ús de codi Java per interpretar alguna de les estructures lèxiques esmentades. L'ús d'estats lèxics està permès. Recordeu que, per aquesta primera PAC, no s'ha de d'implementar cap analitzador sintàctic ni semàntic.

Jocs de proves i plantilla de codi

Juntament amb l'enunciat d'aquesta PAC, s'adjunten diversos jocs de proves: fitxers que contenen, per una entrada esperada, la sortida esperada del vostre analitzador. Durant l'avaluació de la PAC, es tindrà en compte el resultat d'aquests jocs de proves. Cal tenir en compte que l'avaluació d'aquests jocs de proves es **realitzarà de manera automàtica**. Per tant, cal assegurar-se que la sortida de l'analitzador coincideixi amb el contingut dels fitxers de joc de prova. A tal efecte, es proporcionarà una plantilla que us ajudarà a comparar el resultat del vostre analitzador amb els jocs de proves. Per utilitzar correctament la plantilla, cal que tingueu en compte les següents consideracions:

- **No es pot alterar el codi de la plantilla proporcionada**, excepte les parts indicades en un comentari. L'únic fitxer que podeu modificar és el fitxer `Pregunta3.lex`.
- En el codi Java associat a cada acció de l'analitzador lèxic, haureu d'utilitzar la funció `LexerEvaluator.emitToken`. Per exemple, després de reconèixer un enter, haureu de cridar la funció així: `LexerEvaluator.emitToken(TokenType.INTEGER, yytext());`.
- Podeu veure els tipus de token que es poden emetre al fitxer `Eval/TokenType.java`, que són: `IDENTIFIER`, `INTEGER`, `DELIMITER`, `OPERATOR`, `KEYWORD`, `STRING`, `URL`, `PATH`, `COMMENT`[3] i `ERROR`. Els espais en blanc, salts de línia i altres caràcters ignorats no han d'emetre cap token.
- De manera similar, en detectar un error lèxic, caldrà cridar una funció. **S'ha d'emetre un error lèxic per cada caràcter no reconegut de l'entrada amb la següent crida:** `LexerEvaluator.emitUnrecognizedTokenError(yytext());`.
- Les parts de la plantilla que heu de modificar són el codi JLex, a baix de tot, l'implementació de la funció `printExecutionSummary`, al principi del fitxer.
- El punt d'entrada (main) que defineix la plantilla rep dos arguments via línia de comandes. El primer argument és el mode d'execució, que pot ser `run` o `evaluate`. El segon és la ruta d'un fitxer de codi que voleu analitzar, sense la extensió final.

[3] Els comentaris s'inclouen com a tipus de token, només a efectes de millorar la cobertura de l'avaluador. En realitat els comentaris s'han d'ignorar i descarten durant l'anàlisi lèxic, per evitar haver-los de considerar en la gramàtica durant l'anàlisi sintàctic, igual que els salts de línia i espais en blanc.

- Els fitxers de codi **han de tenir obligatòriament l'extensió** `.srv`. Ja que aquesta és l'extensió que espera el programa.
- En mode `run`, s'executarà el vostre analitzador amb el cas de prova indicat. Exemple: `java -jar Yylex run TestCases/case1`
- En mode `evaluate`, s'executarà el vostre analitzador amb el cas de prova indicat, i a més, es comprovarà la sortida amb el cas de prova associat, indicant si l'avaluació és correcta amb el missatge `Success!`, o bé si hi ha hagut algun problema, indicant l'error. Exemple: `java -jar Yylex evaluate TestCases/case1`
- Podeu crear els vostres propis casos de prova, fixeu-vos en el format esperat mirant els fitxers de cas de prova proporcionat.

Solució:

```
import java.io.*;
import java.lang.*;
import java.util.ArrayList;
import Eval.LexerEvaluator;
import Eval.TokenType;

%%
%{

private static int identifierCount = 0;
private static int reservedCount = 0;
private static int numErrors = 0;

public static void printExecutionSummary() {
    System.out.println("Execution Summary:");
    System.out.println(" - Number of identifiers: " + identifierCount);
    System.out.println(" - Number of reserved keywords: " + reservedCount);
    System.out.println(" - Nombre of lexer errors: " + numErrors);
}


/*
  ========================================================
  Below is the main function. It must *NOT* be altered.
  ========================================================
*/
public static void main (String argv[]) throws Exception {

```

```
29          if (argv.length < 2) {
30
31              System.out.println("Wrong number of parameters!");
32              return;
33
34          } else {
35
36              String mode = argv[0];
37              if (!mode.equals("run") && !mode.equals("evaluate")) {
38                  System.err.println("[Arguments Error] The first parameter must be
    either \"run\" or \"evaluate\".");
39              } else {
40                  String caseName = argv[1];
41                  String filePath = caseName + "." + LexerEvaluator.LANGUAGE_EXTENSION;
42                  FileInputStream sourceCodeInputStream = null;   // input file
43
44                  try {
45                      sourceCodeInputStream = new FileInputStream(filePath);
46                  }
47                  catch (FileNotFoundException e) {
48                      System.out.println(filePath + ": Unable to open file");
49                      return;
50                  }
51                  Yylex yy = new Yylex(sourceCodeInputStream);
52                  while (yy.yylex() != -1);
53                  sourceCodeInputStream.close();
54
55                  printExecutionSummary();
56
57                  if (mode.equals("evaluate")) {
58                      LexerEvaluator.evaluateTestCase(caseName + "." + LexerEvaluator.
    TESTCASE_EXTENSION);
59                  }
60              }
61
62          }
63  }
64
65  %}
66
67  %integer
68  %notunix
69
70  NEWLINE=[\n\r]
71  SPACE=[\t ]
72  COMMENT=#.*
73  IDENTIFIERS=[A-Za-z_][A-Za-z0-9_]*
74  INTEGERS=-?[0-9]+
75  STRING=\"[^\"]*\"
76  FILENAME=[\.a-zA-Z0-9_]+
```

```
77  PROTOCOL=(http|https)://
78  DOMAINNAME=[a-zA-Z0-9]+
79  TOPLEVEL=(com|org|net|cat|es|de|fr|it)
80
81  %%
82
83  {COMMENT} {
84      LexerEvaluator.emitToken(TokenType.COMMENT, yytext());
85  }
86
87  /?({FILENAME}/)+{FILENAME} {
88      LexerEvaluator.emitToken(TokenType.PATH, yytext());
89  }
90
91  {PROTOCOL}({DOMAINNAME}.)+{TOPLEVEL}((/{DOMAINNAME})+/?|/)? {
92      LexerEvaluator.emitToken(TokenType.URL, yytext());
93  }
94
95  {STRING} {
96      LexerEvaluator.emitToken(TokenType.STRING, yytext());
97  }
98
99  (audio|switch|json|script|service|plays|opens|closes|int|string|endpoint|request|
        say|let|sleep) {
100     LexerEvaluator.emitToken(TokenType.KEYWORD, yytext());
101     reservedCount++;
102 }
103
104 (=|\+\+|\+|-|\*|/) {
105     LexerEvaluator.emitToken(TokenType.OPERATOR, yytext());
106 }
107
108 (\{|\}|\(|\)|;|:|,) {
109     LexerEvaluator.emitToken(TokenType.DELIMITER, yytext());
110 }
111
112
113 {IDENTIFIERS} {
114     LexerEvaluator.emitToken(TokenType.IDENTIFIER, yytext());
115     identifierCount++;
116 }
117
118 {INTEGERS} {
119     LexerEvaluator.emitToken(TokenType.INTEGER, yytext());
120 }
121
122 {NEWLINE} { }
123 {SPACE} { }
124
125 .  {
```

```
126     numErrors++;
127     LexerEvaluator.emitUnrecognizedTokenError(yytext());
128 }
```

Recursos

Recursos Bàsics

- Mòdul didàctic 1. Visió general
- Mòdul didàctic 2. Anàlisi Lèxic
- Manual de JLex i CUP

Recursos Complementaris

- Material de repàs d'expressions regulars
- Aula de Laboratori de Java de Compiladors

Criteris d'avaluació

La ponderació dels exercicis és la següent:

- Exercici 1: 10%
- Exercici 2: 30%
- Exercici 3: 60%

Pel que fa a l'exercici pràctic de JLex, es valorarà:

- La correcta definició del programa principal que crida l'analitzador lèxic.
- L'ús correcte de les directives JLex.
- Els patrons i accions utilitzats.
- L'ús d'estats lèxics, si és necessari.
- L'ús correcte de la taula de símbols, si s'escau.
- L'assoliment de tots els requisits descrits en l'enunciat.

En concret **es penalitzarà** la implementació de funcionalitats en Java que ja són ofertes per JLex.

Format i data de lliurament

Cal lliurar un document en format PDF, juntament amb el codi, en format `.lex`. Al document pdf cal incloure les respostes a les preguntes teòriques, juntament amb una justificació de les decisions preses a l'exercici pràctic.

Aquest document s'ha de lliurar a l'espai de **Lliurament i Registre d'AC** de l'aula abans de les **23:59 del dia 20 d'octubre de 2020.** **No s'acceptaran lliuraments fora de termini.**