COMPILADOR KSA

Universitat
de les Illes Balears

DIEGO BERMEJO - 45184960B
JOAN BALAGUER - 43219174N
MARC CAÑELLAS - 43478655F
JAUME ADROVER - 43235882E

INDEX

1. Descripció del Llenguatge

El nostre llenguatge es basa en un cos general de programa on hi ha d'haver els subprogrames, les declaracions i les instruccions del programa que es voldrà executar (de l'estil del main de java o c++ o un apartat d'instruccions tipus python). Una peculiaritat del nostre llenguatge és que aquesta funció main principal sempre ha de ser la darrera que aparegui al fitxer. Es a dir, després del tancament de la funció main, no hi pot haver res més. La sintaxi que hem escollit per definir-lo és la següent:

```
main(){
      “mainBody”
}
```

També hem definit una sintaxi per poder escriure subprogrames. Com a conseqüència que el main hagi de ser el darrer procediment descrit al fitxer del programa, qualsevol definició de funcions o procediment s'haurà de fer abans que aquest. La sintaxi que hem escollit per als subprogrames és la següent, tenint en compte que a aquests se li poden passar paràmetres (una per procediments i l'altre per funcions):

```
func "type” "nomFunc"(“args”) {
      “funcBody”
      return "nomvar";
}

proc "nomProc"(“args”) {
      “procBody”
}
```

Quant als tipus de dades que hem implementat ha estat els següents:

- Enter: `int "nomInt"`
- Cadena de caràcters (string): `string "nomString"`
- Booleans: `bool "nom"`

Estructures de dades que podrà implementar pròpies del llenguatge:

- Arrays (únicament implementats per a 1 dimensió):

  ```
  "type" "nom" <- “type”[“valor”];
  ```

  Com es pot observar, en el nostre llenguatge, la declaració d'un array i d'una variable, és la mateixa. Per tant, sempre que es vulgui declarar un array, també s'haurà d'inicialitzar amb la seva dimensió.

Tipus de declaracions amb els tipus de dades i estructures de dades descrites:

- Declaració i inicialització de variables:

  Declaració: `“type” “nomVar”;`

  Declaracio i Inicialitzacio: `“type” “nomVar” <- “valor”;`
- Constants. Quan es declaren, aquestes s'han de inicialitzar i no es podrà modificar el seu valor durant tota l'execució del programa:

```
const “type” “nomConst” <- “valor”;
```

També hem definit els tipus d’operacions que es podrèn dur a terme en el nostre llenguatge. Aquestes són:

- Assignació entre dures variables o entre una variable i un valor:

```
“nomVar” <- “nomVar”;
“nomVar” <- “valor”;
```

- Sentencia condicional per dividir blocs de codi amb una condició booleana:

```
if(“condicio”){
    “ifBody”
}
```

- Selecció múltiple de tipus switch:

```
switch(“nomVar”){
    case “valor”:

    case ”valor”:
}
```

- Sentencies iteratives. N’hem desenvolupat de dos tipus:
  - Bucle while:

```
while (“condicio”){
    “whileBody”
}
```

  - Bucle for:

```
for(“Init”; “condicio”; “asignacio”){
    “forBody”
}
```

Crida a procediments i funcions amb paràmetres. En trobem de dos tipus:

- En cas de voler cridar a una funció, serà obligatori guardar el contingut que retorna aquesta. En cas contrari, saltarà un error:

```
“type” “nomVar” <- "nomFunc" (“args”);
```

- En cas de voler cridar a un pocediment, com que no retorna res es podrà cridar la de la següent forma:

```
“nomProc”(“args”);
```

També hem implementat les expressions aritmètiques i lògiques següents:

- Fent ús de literals del tipus adient:

```
“valorInt” "operador" “valorInt”;
“valorBool” "operador" “valorBool”;
```

- Fent ús de constants i variables

```
“nomVar” "operador" “nomVar”;
```

Em quant a les operacions d’entrada i sortida

- Entrada per teclat:

```
“nomVar” <- input();
```

- Sortida per pantalla:

```
print("valorString");
```

Finalment, aquests són els operadors que hem desenvolupat per al nostre llenguatge:

- Aritmètics: suma, resta, producte, divisió, mòdul

```
+, -, *, /, %
```

- Relacionals: igual, diferent, major, menor, major o igual, menor o igual

```
=, !=, >, <, >=, <=
```

- Lògics: i, o, no

```
and, or, not
```

2. Analitzador Lèxic

Per tal de crear l'analitzador lèxic per al compilador del nostre llenguatge KSA, hem definit els següents tokens (la notació utilitzada per descriure les expressions simples dels tokens, és la mateixa que s'utilitza al fitxer .flex del codi):

Declaracions

```
- id = [A-Za-z_][A-Za-z0-9_]*
```

Un identificador sempre haurà de començar amb una lletra i després d'aquesta primera lletra obligatòria podran venir tantes lletres i nombre com desitjo el programador.

```
- integer = {sub}?[0-9][0-9]*
```

Un enter el definim com un primer caràcter opcional per poder definir els nombres negatius (-) continuat d'un primer caràcter numèric que sempre hi serà seguit de tants caracters numèrics com el programador vulgui definir.

```
- str = [\"](([A-Za-z0-9_)]) | ({blank}))*[\"]
```

Finalment, definim una cadena de caràcters (string) com una cadena de tants de caràcters (tant numèrics com alfabètics) com programador desitgi, entre cometes dobles (“). A més, també es possible que alguns d'aquests caracters sigui un caracter blanc (espai). Per aquest motiu posem una or amb el caràcter blanc , ja que si el programador ho desitja, pot inserir espais dins l'string.

Símbols Operadors

```
- bg      = \>
- sm      = \<
- bg_eq   = >=
- sm_eq   = <=
- eq      = =
- neq     = \!=
```

```
- add          = \+
- sub          = \-
- mul          = \*
- div          = \/
- mod          = \%
- asig         = <-
- smcol        = ;
- twodots      = :
- com          = ,
- brop         = \{
- brcl         = \}
- lop          = \[
- lclose       = \]
- lparen       = \(
- rparen       = \)
```

Amb aquests tokens podrem reconèixer els distints operadors que posem a disposició del programador per poder fer comparacions i assignacions lògiques.

Paraules Reservades

```
- r_if        = if
- r_else      = else
- r_switch    = switch
- r_case      = case
- r_default   = default
- r_function  = func
- r_procedure = proc
- r_return    = return
- r_while     = while
- r_for       = for
- r_const     = const
- r_and       = and
- r_or        = or
- r_not       = not
- r_int       = int
- r_bool      = bool
- r_main      = main
- r_print     = print
- r_input     = input
- r_string    = string
- r_array     = array
```

Amb aquests tokens, podrem reconèixer totes les paraules reservades amb les quals el programador podrà realitzar les funcions, bucles, condicionals, etc. En definitiva, crear l'estructura d'un programa del llenguatge KSA.

Altres tokens

- `new_line = ([\n\r]|(\n\r))+`
- `blank = [ \t\r]+`
- `r_comment = "//"([^\n])*`

Finalment, hem definit un token per poder reconèixer els salts de linea que vulgui definir el programador, a més del caràcter en blanc i els comentaris.

En cas que el compilador no hagi reconegut cap dels altres tokens, reconeixerà el token error. Amb aquest, tractarem els errors lèxics, que sorgeixen quan el programador insereix un caràcter que no coincideix amb cap token en el programa.

3. Analitzador Sintàctic

Pel que fa a l'analitzador sintàctic, hem desenvolupat una gramàtica per poder generar programes del llenguatge KSA. Aquesta l'hem generada amb l'eina CUP que se'ns va proporcionar durant el curs. Hem utilitzat aquesta eina, ja que és la que més a mà teníem per poder generar la gramàtica del nostre llenguatge. Per aquest motiu, la sintaxi que utilitzarem al document per mostrar la gramàtica, és la mateixa que la que utilitzem al document .cup que s'anomena `sintactic_og.cup`.

Primerament, anomenarem els principals elements terminals de la nostra gramàtica (que es rebran de l'analitzador lèxic de l'apartat anterior del document):

Identificadors

- `terminal id,bool,str;`

Enters

- `terminal integer;`

Operadors

- `terminal bg, sm, bg_eq, eq, neq;`

Operacions

- `terminal add, sub, mul, div, mod;`

Caràcters Suplementaris

- `terminal asig, smcol, twodots, com, brop, brcl, lop, lclose, lparen, rparen;`

Paraules Reservades

- `terminal r_if, r_else, r_switch, r_case, r_default, r_function, r_procedure, r_return, r_while, r_for, r_const, r_and, r_or, r_not, r_int, r_bool, r_main, r_string, r_input, r_print, r_array;`

Per altra banda, hem definit els elements no terminals que pertanyen a la nostra gramàtica. Aquests, els hem definit com a objectes d'una classe en concret. Per exemple, l'element no terminal `ifStatement`, és un objecte de la classe `SymbolIfStatement`. Aquesta, contindrà tots els atributs i mètodes necessaris per a poder definir i gestionar una sentència de tipus if. A més, també contindrà altres mètodes que ens ajudaran en fases més avançades del compilador com l'anàlisi semàntica (comprovació de tipus) i la generació de codi intermedi. Cada element no terminal té definit la seva pròpia classe i aquestes són les següents:

```
- non terminal  SymbolProgram   program;
- non terminal  SymbolDeclList  declList;
- non terminal  SymbolDecl   decl;
- non terminal  SymbolVarDecl  varDecl;
- non terminal  SymbolVarInit  varInit;
- non terminal  SymbolFuncDecl   funcDecl;
- non terminal  SymbolFuncCap   funcCap;
- non terminal  SymbolProcDecl   procDecl;
- non terminal  SymbolType   type;
- non terminal  SymbolStatementList   statementList;
- non terminal  SymbolStatement   statement;
- non terminal  SymbolWhileStatement   whileStatement;
- non terminal  SymbolForIteration   forIteration;
- non terminal  SymbolForInit   forInit;
- non terminal  SymbolForPostExpression   forPostExpression;
- non terminal  SymbolIfStatement   ifStatement;
- non terminal  SymbolExpressioSimple  exprSimple;
- non terminal  SymbolSubProgramCall   subProgramCall;
- non terminal  SymbolSubProgramContCall   subProgramContCall;
- non terminal  SymbolArray      array;
- non terminal  SymbolArrayInit  arrayInit;
- non terminal  SymbolContCap ContCap;
- non terminal  SymbolReturn return;
- non terminal  SymbolOp op;
- non terminal  SymbolOperacio operacio;
- non terminal  SymbolValor valor;
- non terminal  SymbolBoolOp boolOp;
- non terminal  SymbolAritOp aritOp;
```

```
- non terminal  SymbolRealOp realOp;
- non terminal  SymbolSwitchCase SwitchCase;
- non terminal  SymbolLCases LCases;
- non terminal  SymbolCase Case;
- non terminal  SymbolDefault Default;
- non terminal  SymbolInputStatement inputStatement;
- non terminal  SymbolPrintStatement printStatement;
- non terminal  SymbolLiteral literal;
- non terminal  SymbolArr     pars;
```

A més, també hem definit un tipus de precedència específica per a alguns dels terminals:

```
- precedence right asig;
- precedence left  add, sub;
- precedence left  mul, div, mod;
```

Una vegada definides les parts de les quals es compon la nostra gramàtica, és hora de definir les produccions d'aquesta. Aquesta és la següent (cada guió representa les produccions associades a un sol element no terminal):

```
- program ::= declList r_main lparen rparen brop statementList
  brcl;

- declList ::= declList decl
               |
               ;

- decl ::= varDecl smcol
          |funcDecl
          |procDecl
          ;

- varDecl ::= r_const type id varInit
             |type id varInit
             ;

- varInit ::= asig exprSimple
             |asig arrayInit
             |
             ;

- arrayInit ::= type lop integer lclose
                ;

- type ::= r_int
          |r_bool
```

```
          |r_string
          ;

- return   ::= r_return exprSimple smcol
               ;
- funcDecl ::= r_function type id funcCap brop statementList
  return brcl;

- funcCap ::= ContCap rparen
             |lparen rparen
             ;

- procDecl ::= r_procedure id funcCap brop statementList brcl
              ;

- ContCap ::= lparen type pars id
             | ContCap com  type pars id
             ;

- pars ::= r_array
          |
          ;

- statementList ::= statement statementList
                   |
                   ;

- statement ::= varDecl smcol
               |exprSimple smcol
               |forIteration
               |whileStatement
               |printStatement smcol
               |ifStatement
               |SwitchCase
               ;

- inputStatement ::= r_input lparen rparen
                    ;

- literal ::= str:str
             |id
             ;

- exprSimple ::= valor operacio
                ;

- valor::= id
          |array
```

```
          |integer
          |bool
          |r_not lparen exprSimple rparen
          |subProgramCall
          |lparen exprSimple rparen
          |inputStatement
          |str:str
          ;

- array::= id lop exprSimple lclose
          ;

- operacio::=op exprSimple
             |varInit
             ;

- op::= boolOp
      |aritOp
      |realOp
      ;

- boolOp::= r_and
          |r_or
          ;

- aritOp ::= add
            |sub
            |mul
            |div
            |mod
            ;

- realOp::= neq
          |eq
          |bg
          |sm
          |bg_eq
          |sm_eq
          ;

- subProgramCall ::= id lparen rpare
                    |id lparen subProgramContCall rparen
                    ;
- subProgramContCall ::= valor
                        |subProgramContCall com valor
                        ;
```

- SwitchCase ::= r_switch lparen exprSimple rparen brop LCases Default brcl;

- LCases ::= Case LCases
  |
  ;

- Case ::= r_case exprSimple twodots statementList;

- Default ::= r_default twodots statementList:state
  |
  ;

- forIteration ::= r_for lparen forInit smcol exprSimple smcol forPostExpression rparen brop statementList brcl;

- forInit ::= varDecl
  |
  ;

- forPostExpression ::= exprSimple
  |
  ;

- whileStatement ::= r_while lparen exprSimple rparen brop statementList brcl;

- ifStatement ::= r_if lparen exprSimple:expr rparen brop statementList brcl
  |r_if lparen exprSimple rparen brop statementList brcl r_else brop statementList brcl
  ;

Amb aquesta gramàtica, és possible generar programes del llenguatge KSA. Per tant, el nostre compilador podrà reconèixer les estructures pròpies del nostre llenguatge una vegada l'analitzador lèxic hagi fet les identificacions dels diversos tokens que es trobin al programa. A mesura que els vagi identificant, els anirà enviant a l'analitzador sintàctic, que anirà comprovant la correctesa en conjunt dels tokens que li van arribant. Per qüestions de visualització, en el document només hem mostrat les produccions i terminals, però en el fitxer anomenat "sintactic_og.cup", també hi apareixen altres elements que aquí no apareixen. Alguns d'aquests són noms de variables per poder passar el contingut d'una producció a la corresponent classe de la mateixa o la gestió d'errors sintàctics (es troba al programa una estructura que no concorda amb cap producció de la gramàtica).

4. Analitzador Semàntic

Per a realitzar correctament la comprovació dels tipus a les diferents produccions hem decidit crear una classe Semàntic.java. L'altra opció era comprovar el tipus a la vegada que generem el codi. Hem decidit no fer-ho així per a poder tenir una estructura més clara.

Cridem al mètode de comprovació de tipus i en cas que sigui correcte generem el codi.

Nom Mètode	Funcionalitat
gestExpr()	Comprovar que una expressió és correcta, és a dir, que el seu TipusSub és el que s'esperava i no és null (hi ha errors dins l'expressió simple).
gestPrint()	Comprovar que la variable dins un print sigui string
isExprCorrecta()	Comprova que una expressió sigui correcta
gestFunc()	Veure que una funció retorna el que ha de retornar, per exemple: una funció boolean no pot retornar un int.
gestAsigDecl()	Mètode per a veure si una assignació d'una declaració és correcte
gestAsig()	Mètode per a veure si una assignació es correcta
gestSwitch()	Mètode per a veure que les cases tenen el mateix tipus subjacent que l'expressió
gestForLoop()	Mètode per a veure si un bucle for és correcte, comprovant la inicialització, la condició intermèdia i la post expressió.
gestIdArray()	Mètode per a veure si

5. Taula de Símbols

La taula de símbols és una estructura implementada a la classe `TaulaSimbols.java`. Aquesta consisteix en una combinació dels següents elements:

- `ArrayList<Simbol>`: es van afegint en aquesta llista tots els símbols, de manera que tindrem totes les variables i procediments a dintre.
- HashMap per a tots els àmbits `<Integer,String>` on la clau és el nivell i el valor és el nom de l'àmbit.

Per entendre millor com funciona veurem un exemple:

Taula Símbols(ArrayList<>): a,b1,b2
b,c,d.
Taula d'àmbits (HashMap<>): 1:a ,2:b

Imaginem que tenim dos procediments a,b. Si volem ubicar les variables de a haurem de començar a mirar la llista a partir de la posició a, fins al màx de variables dins aquell àmbit.

Hem de tenir present que la classe Símbol conté els atributs que fan referència al camp **descripció** explicat a la teoria.

Funcionalitats implementades:

- `afegeixSimbol`: mètode que serveix per a afegir un símbol a partir del seu identificador, tipus subjacent, tipus, posició i dimensió. Si és una funció també s'afegeix a la taula d'àmbits, que en el nostre cas és el HashMap.

- `afegeixNivell`: aquest mètode incrementa el comptador de nivells Màxim i assigna a nivell actual el nombre corresponent. Per exemple: si cream una tercera funció, el nivell d'aquesta sirà el 3.

- `consultaFunc`: mètode que retorna l'objece `Simbol` a partir d'un identificador que li passem per paràmetre. Aquesta consulta la taula d'àmbits.

- `consultaSimbol`: mètode que retorna un Símbol (o 'null' si no el troba) a partir d'un identificador passat per paràmetre. Aquest mètode recorre la taula de símbols comparant l'element amb el passat per paràmetre.

- `consultaParametre`: mètode que retorna un Símbol a partir d'un altre símbol passat per paràmetre i un valor enter.

- `nParametresFunc`: mètode que retorna el nombre de paràmetres d'una funció que li passem per paràmetre (li passem un objecte de tipus `Simbol`) .

6. Generació de Codi Intermedi

Una vegada hem comprovat que el programa que se li ha passat al compilador és correcte lèxicament, sintàcticament i semànticament correcte, podem passar a la generació de codi intermedi. Aquest, es genera per a facilitar la tasca de crear el codi assemblador. Per a la generació d'aquest, tenim una classe principal anomenada `codiTresAdreces.java`. Conté els atributs i mètodes necessaris per a generació d'instruccions de 3 adreces i afegir-les a l'atribut codi d'aquesta. Aquesta classe s'ajuda d'altres classes:

Nom Classe	Funcionalitat
`Etiqueta.java`	Conté els atributs i mètodes necessaris per descriure una etiqueta per fer els salts al codi de tres adreces.
`Instruccio.java`	Conté els atributs i mètodes necessaris per descriure una instrucció del codi de tres adreces. Aquest és el tipus de dada que s'emmagatzemarà a l'arraylist codi de la classe principal `codiTresAdreces`. Algunes d'elles tenen una certa posibilitat de tipus de destins. Per exemple NE i EQ ('not equal', 'equal'), poden tenir com a destí una etiqueta(on es fa el bot després d'avaluar la condició), o una variable temporal, on s'assigna el resultat de l'avaluació
`Operand.java`	Conté els atributs i mètodes necessaris per descriure un operand que s'utilitzaran per poder crear els objectes de tipus `Instruccio`.
`OperandsCTA.java`	Es tracta d'una classe que conté un enumerat. Aquest enumerat conté els tipus d'operands que podrem trobar a les instruccions del codi de tres adreces
`Parametre.java`	Conté els atributs i mètodes necessaris per descriure un paràmetre d'una funció que s'hagi declarat o cridat al programa original.
`Procediment.java`	Conté els atributs i mètodes necessaris per descriure un procediment per tal de poder inserir-lo al codi de tres adreces.
`Taulaprocediments.java`	Conté els atributs i mètodes necessaris per descriure la taula que conté tots els procediments que s'han definit al programa que ha fet el programador.
`TaulaVariables.java`	Conté els atributs i mètodes necessaris per descriure la taula que conté totes les variables que s'han definit al programa que ha fet el programador i les variables temporals que anem generant a mesura que generem les distintes instruccions de codi tres adreces.
`TipusInstruccionsCTA.java`	Es tracta d'una classe que conté un enumerat. Aquest

	descriu el tipus d'instruccions que ens podrem trobar al codi de tres adreces.
`Variable.java`	Conté els atributs i mètodes necessaris per descriure una variable.

7. Generació de Codi Assemblador

Una vegada hem generat el codi intermedi amb les classes de l'apartat anterior, hem creat una classe anomenada `codi68k.java`, la qual, combinada amb la classe `Writter.java` (utilitzada per escriure les instruccions en assemblador dins un fitxer), aconseguim generar el codi assemblador del programa que el programador ha escrit en el llenguatge KSA. En el nostre cas, hem decidit produir el codi assemblador per a l'arquitectura del processador Motorola 68000. Hem decidit fer-ho per aquesta arquitectura en concret pel fet que era l'arquitectura amb la qual vàrem aprendre a programar en llenguatge assemblador a les assignatures Estructura de Computadors 1 i 2. A més, per a aquest tipus d'arquitectura existeix un simulador anomenat "easy68k" amb el qual hem pogut provar els programes en llenguatge assemblador sobre aquesta arquitectura. Per tant, a la classe `codi68k.java` hi trobarem tots els mètodes necessaris per a la traducció d'instruccions de codi tres adreces a instruccions del llenguatge assemblador de l'arquitectura Motorola 68000. No hem utilitzat cap llibreria externa per al codi assemblador.

8. Guia per a l'execució del compilador

En aquest apartat del document volem facilitar al programador la tasca d'execució del nostre compilador. Primerament, començarem anomenant les dependències que seran necessàries tenir instal·lades a l'ordinador per no tenir problemes durant l'execució:

- Serà necessari tenir instal·lat java amb una versió igual o superior a la 19.0.1 (aquesta és la versió amb la qual vàrem crear el projecte NetBeans de la pràctica).
- No serà necessària la instal·lació ni de flex ni de cup (llibreries utilitzades a l'analitzador lèxic i sintàctic), ja que hem inserit dins el mateix projecte NetBeans una carpeta anomenada "lib" amb aquestes dependències.
- Una vegada s'hagi generat el codi assemblador, per poder-lo executar serà necessari tenir instal·lat el simulador del Motorola 68000. Aquest s'anomena "easy68k". L'enllaç de descàrrega el pots trobar aquí.

8.1 Recompilar gramàtica/lèxic (Opcional)

Una vegada tenim instal·lades totes les dependències, podem passar explicar com es posa en marxa el nostre compilador.

1. Primer de tot, s'ha d'escriure un programa en el llenguatge KSA. Per tant, creeu un fitxer (que podeu anomenar com vulgueu) amb l'extensió .ksa i escriviu un programa en

aquest llenguatge. Aquest programa el podeu emmagatzemar en el directori que vulgueu.

2. Una vegada tenim el programa crear, hem de posar en marxa el compilador. Per fer-ho, obrim la nostra terminal i ens situem al directori del projecte "KSA_Compiler". Una vegada hi som, executa'm la següent comanda:

   ```
   cd src/compiler/
   ```

3. Una vegada ens trobem en aquest directori, podrem veure que hi ha un fitxer anomenat "execute.sh". Aquest script condensa totes les comandes necessàries per executar tant l'analitzador lèxic com el sintàctic. En cas de poder-lo utilitzar (en cas de tenir un SO Linux o Mac OS) es pot executar mitjançant la següent comanda:

   ```
   sh execute.sh
   ```

4. En cas de no poder-lo utilitzar per estar en un sistema operatiu com Windows, es poden executar les mateixes comandes que s'executen a l'script però de forma individual. Les que s'utilitzen per poder executar el lèxic son les següents:

   ```
   cd lexic

   java -jar jflex-1.6.1.jar lexic.flex

   cd ..

   cd sintactic

   java -jar-cup-11b.jar sintactic_og.cup
   ```

8.2 Executar fitxers

Per a poder executar fitxers amb el nostre compilador cal ubicar-se amb la terminal a la carpeta `dist`. A continuació, escriurem la següent comanda:

```
java -jar KSA.jar path/to/file
```

On `path/to/file` sirà el camí des de la carpeta `dist` fins al fitxer a executar.

Exemples d'execució: [vídeo explicatiu de les millores realitzades (convocatòria extraordinària)]

Quan s'hagi executat, es generaran els següents fitxers a la carpeta arrel del projecte:

- `NOMFITXER_tokens.txt`: conté tots els tokens que ha pogut identificar l'analitzador lèxic.
- `NOMFITXER_errors.txt`: conté tots els errors que s'hagin detectat durant el proces de compilació. Aquests indicaran el tipus d'error detectat (lèxic, sintactic o

semàntic). A més, aquests errors també apareixeràn en vermell a la finestra de surtida una vegada s'executi el main. Per evitar errors de recursivitat, la consola únicament mostra el primer error de la llista generada.

- `NOMFITXER_TaulaSimbols.txt`: conté el contingut de la taula de simbols mostrat en un fitxer per poder comprovar la seva correctesa.
- `NOMFITXER_codiIntermedi.txt`: conté totes i cada una de les instruccions de codi tres adreces que s'han generat una vegada el compilador ha comprovat que el programa d'entrada és lèxicament, sintàcticament i semànticament correcte.
- `NOMFITXER_TaulaVariables.txt`: conté el contingut de la taula de variables mostrat en un fitxer per poder comprovar la seva correctesa.
- `NOMFITXER_TaulaProcediments.txt`: conté el contingut de la taula de procediments mostrat en un fitxer per poder comprovar la seva correctesa.
- `NOMFITXER_exe.68K`: conté el codi assemblador generat a partir del codi tres adreces.

`NOMFITXER` es refereix al nom del fitxer que conté el codi compilat en el llenguatge KSA.

Link del video explicatiu de la pràctica (convocatoria ordinaria)