DARTAM

Universitat
de les Illes Balears

Integrantes: Juan Arturo Abaurrea Calafell, Daniel Salanova Dmitriyev, Marta González Juan
Asignatura: 21780 - Compiladores
Profesor: Pedro Antonio Palmer Rodríguez

Índice

Introducción:

Como objetivo final de la asignatura de compiladores se busca desarrollar un compilador desde 0, con el que podamos pasar de un lenguaje inventado por el equipo de trabajo y que este sea tratado, con tal de generar un ejecutable en ensamblador que funcione de manera correcta.

Nuestro lenguaje, DARTAM, es un lenguaje fuertemente tipado y con una estructura muy similar a la que podríamos encontrar en Java.

El proyecto se ha dividido en 6 partes: análisis léxico, análisis sintáctico, análisis semántico, generación de código intermedio, optimizaciones y generación de código máquina. Cada una de las partes tiene un objetivo específico, de tal manera se verá reflejado en el propio proyecto en forma de diferentes ficheros.

Hemos usado JFlex para llevar a cabo el análisis léxico, mientras que para el sintáctico hemos usado CUP. En cambio, para el análisis semántico, hemos generado a mano el código necesario con tal de poder gestionar cada una de las expresiones. Finalmente, todas las instrucciones en lenguaje ensamblador están hechas en 68K, por lo que se necesitará un programa externo tal como easy68K con tal de poder ejecutar el producto final y comprobar su funcionamiento.

A continuación, se verá en cada uno de los diferentes apartados del documento, una explicación de cada una de las diferentes fases implementadas del proyecto, junto con las restricciones del lenguaje, con tal de poder dar una visión general de nuestro compilador.

Análisis Léxico:

En este apartado se mostrarán los diferentes tokens obtenidos del desarrollo de nuestro lenguaje, junto a sus patrones. De esta manera, todo aquello dentro de los patrones generados será tratado por el lenguaje, cualquier cosa fuera de eso originará un error.

Reutilizables
• sub_digit = [0-9] • sub_letra = [A-Za-z] // no confundir con carácter • sub_car = {sub_digit}|{sub_letra}

Estos serán los tokens que se usarán en combinación, con tal de generar los diferentes valores del lenguaje, identificadores...

Símbolos
• sym_parenIzq = \(• sym_parenDer = \) • sym_llaveIzq = \{ • sym_llaveDer = \} • sym_bracketIzq = \[• sym_bracketDer = \] • sym_endInstr = \; • sym_coma = \, • sym_comillaS = \' • sym_comillaD = \" • sym_punto = \.

Los símbolos incluyen, símbolos propiamente dichos, que se usarán para la creación de funciones, tuplas, arrays, string o char (en caso de comillas).

Operadores
• op_eq = \= • op_diferent = \/\= • op_mayor = \> • op_menor = \< • op_mayorEq = (\>\=)|(\=\>) • op_menorEq = (\<\=)|(\=\<) • op_inc = \+\+ • op_dec = \-\- • op_sum = \+ • op_res = \- • op_mul = \ • op_div = \/ • op_pot = ** • op_porcent = \%

- op_swap = \<\>
- op_or = \|
- op_and = &
- op_mod = \\
- op_neg = \¬
- op_asig = \:
- op_and_asig = \&\:
- op_or_asig = \|\:
- op_sum_asig = \+\:
- op_res_asig = \-\:
- op_mul_asig = *\:
- op_div_asig = \/\:
- op_pot_asig = **\:
- op_mod_asig = \\\:
- op_cond = \?
- op_arrow = \-\>

Todos los operadores que vemos aquí nos permiten hacer las siguientes operaciones:

- Aritméticas: suma, resta, división, multiplicación, porcentaje, módulo, potencia.
- Condicionales: or, and, not, eq, ne, lt, gt, le, ge.
- Asignaciones + Aritméticas: En el lenguaje se permite hacer una asignación a una variable y al mismo tiempo hacer una operación aritmética con dicha variable. Estas operaciones son tales como incrementos y decrementos(a++, a--) o por ejemplo operaciones del estilo (a +: b) siendo esto lo mismo que (a : a + b). Estas últimas operaciones se pueden hacer con cualquier tipo de operador aritmético.

Tipos

- type_char = "car"
- type_int = "ent"
- type_bool = "prop"
- type_void = "vacio"

Los tipos que nuestro lenguaje acepta son los siguientes: Carácteres, Enteros, Booleanos, Strings (como arrays de carácteres) y Null o Void.

Palabras reservadas

- kw_const = "inmut" // inmutable
- kw_main = "inicio"
- kw_method = "f"
- kw_if = "si"
- kw_elif = "sino"
- kw_else = "no"
- kw_switch = "select"

- kw_case = "caso"
- kw_default = "_"
- kw_while = "loop"
- kw_continue = "continuar"
- kw_break = "parar"
- kw_return = "pop"
- kw_true = "cierto"
- kw_false = "falso"
- kw_in = "scan"
- kw_out = "show"
- kw_read = "from"
- kw_write = "into"
- kw_tuple = "tupla"

Tendremos que crear un conjunto de tokens con palabras reservadas con tal de evitar problemas a la hora de ejecutar las siguientes fases. Las variables por ejemplo no se podrán llamar como dichos valores. En este punto podemos observar que tendremos instrucciones tales como bucles, if, switch, operaciones de entrada y salida como pueden ser from y into.

Valores

- val_decimal = {sub_digit}+
- val_binario = 0b[01]+
- val_octal = 0o[0-7]+
- val_hex = 0x[A-Fa-f0-9]+
- val_prop = {kw_true}|{kw_false}
- val_char = "'"[^"']"'"
- char_vacio = "''"
- char_string = "'"[^"'][^"']+"'"
- val_cadena = "\""[^"\""]*"\""
- id = ({sub_letra}|_)({sub_car}|_)*

Aquí podemos observar las reglas relacionadas con los patrones que pueden tener los diferentes valores de cada uno de los tipos.

Casos especiales

- espacioBlanco = [\t]+
- finLinea = [\r\n]+
- comentario = ("\/\/".*)|("#"[^"#"]*"#")

Como carácteres especiales, y aquellos que no trataremos para el análisis sintáctico ni semántico encontramos espacios en blanco, finales de líneas y comentarios. Cabe aclarar que con los comentarios dejaremos hacer comentarios monolínea y multilínea.

Podremos observar de manera completa todas las reglas dentro del fichero Scanner.flex adjuntado con el proyecto.

Estos serán el tratamiento de ciertos errores que se pueden generar con el uso de las comillas simples y comillas dobles .

```
{sym_comillaS}
{sym_comillaD}
{char_vacio}
{char_string}
```

Los siguientes casos que se pueden encontrar dentro del fichero son los siguientes:

- {sym_comillaS}: Error que se indica cuando solo hay una comilla simple y esta no se cierra.
- {sym_comillaD}: Error que se ocasiona cuando una comilla doble no se cierra y queda únicamente una.
- {char_vacio}: Error originado cuando un elemento de tipo char se incorpora como ‘’, sin nada dentro de las comillas, lo que origina un error. Los elementos de tipo char deben incluir un carácter entre sus comillas
- {char_string}: Error creado debido a que un char contiene más de un elemento entre sus comillas simples, por lo que no sería correcto.

El siguiente método usado durante el análisis léxico es *procesarNumero(),* este permite que dado un número categorizarlo en base al carácter que tiene delante. A Su vez *procesarNumero()*, permite devolver un error en caso de que el número sea menor que Integer.MIN y dado un valor superior a Integer.MAX.

```
private Symbol procesarNumero() {
  Integer numero;
  try {
    if (yytext().length()>1) {
      int base;
      switch (yytext().charAt(1)) {
          case 'b' -> { base = 2; }
          case 'o' -> { base = 8; }
          case 'x' -> { base = 16; }
          default -> { base = 10; }
      }
      if (base == 10) {
        numero = Integer.parseInt(yytext(), base);
      } else {
```

```
        numero = Integer.parseInt(yytext().substring(2), base);
      }
    } else {
      numero = Integer.parseInt(yytext());
    }
    return symbol(ParserSym.ENT, numero);
  } catch (Exception e) { /* Error inesperado */
    errores += errorToString();
    errores += "No se permiten numeros fuera del rango " + Integer.MIN_VALUE +
"..." +Integer.MAX_VALUE+"\n";
    return symbol(ParserSym.error);
  }
}
```

Análisis Sintáctico:

En esta fase se ha optado por usar CUP como herramienta para poder generar un parser con tal de llevar a cabo un análisis LALR para nuestra gramática.

- **Gramática**

```
SCRIPT -> SCRIPT_ELEMENTO SCRIPT
         | MAIN

SCRIPT_ELEMENTO -> KW_METHOD TIPO_RETORNO ID LPAREN PARAMS RPAREN LKEY BODY RKEY
         | DECS
         | TUPLE ID LKEY MIEMBROS_TUPLA RKEY

TIPO_RETORNO -> TIPO
         | VOID
MIEMBROS_TUPLA -> DECS MIEMBROS_TUPLA
         | λ

MAIN -> KW_METHOD VOID KW_MAIN LPAREN RPAREN LKEY BODY RKEY
         | MAIN SCRIPT_ELEMENTO

BODY -> METODO_ELEMENTO:et1 BODY
        | λ

TIPO  -> TIPO_PRIMITIVO
        | TIPO_PRIMITIVO DIMENSIONES
        | TIPO_PRIMITIVO DIMENSIONES_VACIAS
        | TUPLE ID
        | TUPLE ID DIMENSIONES
        | TUPLE ID DIMENSIONES_VACIAS

TIPO_PRIMITIVO  -> KW_BOOL
        | KW_INT
        | KW_CHAR
        |      λ

PARAMSLISTA  -> TIPO ID COMMA PARAMSLISTA
        | TIPO ID

DECS -> KW_CONST TIPO DEC_ASIG_LISTA ENDINSTR
        | TIPO DEC_ASIG_LISTA ENDINSTR

DIMENSIONES  -> LBRACKET ATOMIC_EXPRESSION RBRACKET DIMENSIONES
                 | LBRACKET ATOMIC_EXPRESSION RBRACKET

DIMENSIONES_VACIAS -> LBRACKET RBRACKET DIMENSIONES_VACIAS
                       | LBRACKET RBRACKET



DEC_ASIG_LISTA  -> ID ASIG_BASICO COMMA DEC_ASIG_LISTA
        | ID ASIG_BASICO

ASIG_BASICO -> AS_ASSIGN OPERAND
        | λ
```

```
METODO_ELEMENTO -> INSTR
        | LOOP
        | IF
        | SWITCH

INSTR -> FCALL ENDINSTR
        | RETURN
        | DECS
        | ASIGS ENDINSTR
        | SWAP
        | KW_CONTINUE ENDINSTR
        | KW_BREAK ENDINSTR

FCALL  -> METODO_NOMBRE LPAREN OPERANDS_LISTA RPAREN
        | METODO_NOMBRE LPAREN RPAREN

METODO_NOMBRE -> ID
        | SCAN
        | SHOW
        | INTO
        | FROM

OPERANDS_LISTA -> OPERAND COMMA OPERANDS_LISTA
        | OPERAND

RETURN -> KW_RETURN ENDINSTR
        | KW_RETURN OPERAND ENDINSTR

SWAP -> ID OP_SWAP ID ENDINSTR

ASIGS  -> ASIG COMMA ASIGS
        | ASIG

ASIG  -> ID ASIG_OP OPERAND
        | ID DIMENSIONES AS_ASSIGN OPERAND
        | ID OP_MEMBER ID AS_ASSIGN OPERAND
        | ID OP_INC
        | ID OP_DEC
        | OP_INC ID
        | OP_DEC ID

ASIG_OP -> AS_ASSIGN
        | AS_ADDA
        | AS_SUBA
        | AS_MULA
        | AS_DIVA
        | AS_POTA
        | AS_MODA
        | AS_ANDA
        | AS_ORA

OPERAND -> ATOMIC_EXPRESSION
        | FCALL
        | LPAREN OPERAND RPAREN
        | UNARY_EXPRESSION
        | BINARY_EXPRESSION
```

```
        | CONDITIONAL_EXPRESSION
        | OPERAND DIMENSIONES
        | OPERAND OP_MEMBER ID
        | LPAREN TIPO_PRIMITIVO RPAREN OPERAND
        | LPAREN KW_CHAR LBRACKET RBRACKET RPARENOPERAND


UNARY_EXPRESSION-> L_UNARY_OPERATOR:et1 OPERAND
        | OPERAND R_UNARY_OPERATOR

L_UNARY_OPERATOR ->OP_NOT
        | OP_INC
        | OP_DEC
        | OP_AD
        | OP_SUB


R_UNARY_OPERATOR -> OP_PCT
        | OP_INC
        | OP_DEC

BINARY_EXPRESSION -> OPERAND BINARY_OPERATOR OPERAND

CONDITIONAL_EXPRESSION -> OPERAND OP_COND OPERAND ARROW OPERAND


ATOMIC_EXPRESSION -> ID
        | STRING
        | PROP
        | ENT
        | CAR

BINARY_OPERATOR -> OP_ADD
        | OP_SUB
        | OP_MUL
        | OP_DIV
        | OP_MOD
        | OP_POT
        | OP_EQ
        | OP_BEQ
        | OP_BT
        | OP_LEQ
        | OP_LT
        | OP_NEQ
        | OP_AND
        | OP_OR

LOOP  -> KW_LOOP LOOP_COND LKEY BODY RKEY
        | KW_LOOP LKEY BODY RKEY LOOP_COND ENDINSTR

PAREN_LOOP_COND -> LPAREN PAREN_LOOP_COND RPAREN
        | DECS OPERAND ENDINSTR ASIGS

LOOP_COND  -> OPERAND
        | PAREN_LOOP_COND

IF  ->  KW_IF OPERAND LKEY BODY RKEY ELIF ELSE
```

```
ELIFS ->  ELIF ELIFS
        | λ

ELIF ->  KW_ELIF OPERAND LKEY BODY RKEY

ELSE ->  KW_ELSE LKEY BODY RKEY
        | λ

SWITCH ->  KW_SWITCH OPERAND LKEY CASO PRED3 RKEY

CASO ->  CASO KW_CASE OPERAND ARROW LKEY BODY RKEY
        | λ

PRED ->  KW_CASE KW_DEFAULT ARROW LKEY BODY RKEY
        | λ
```

- **Métodos del analizador**

En cuanto a los métodos que se usan para el análisis sintáctico encontramos:

```
init with {:
    anterior = new ArrayList<>();
    anterior.add(null);
    anterior.add(null);
    anterior.add(null);
:};

scan with {:
    ComplexSymbol s = (ComplexSymbol) getScanner().next_token();
    anterior.add(0, s);
    anterior.remove(3);
    return s;
:};

parser code {:
    private ArrayList<ComplexSymbol> anterior;
    private String errores = "";
    public String getErrores() {
        return errores;
    }
    /**
     * Error cuando no es posible una recuperacion de errores.
    **/
    @Override
```

```
    public void unrecovered_syntax_error(Symbol cur_token) {
        String causa = "" + cur_token.value;
        if (cur_token.sym == ParserSym.EOF) {
            causa = "No se ha encontrado metodo main. Sintaxis: \n"+
                "f void inicio(){ # codigo # }\n";
        }
        errores += "No se ha podido recuperar del ultimo error. \nCausa: " +
causa;
        done_parsing();
    }

    /**
     * Error sintactico.
    **/
    @Override
    public void syntax_error(Symbol cur_token){
        report_error("Error sintactico: ", cur_token);
    }

    @Override
    public void report_error(String message, Object info) {
        if (cur_token.sym == ParserSym.EOF) {
            return;
        }
        boolean englishChar = false, englishInt = false;
        if (anterior.size() > 2 && anterior.get(2) != null) {
            englishChar = anterior.get(2).value.toString().equals("char");
            englishInt = anterior.get(2).value.toString().equals("int");
        }
        if (!englishChar && !englishInt && anterior.size() > 1 &&
anterior.get(1) != null) {
            englishChar = anterior.get(1).value.toString().equals("char");
            englishInt = anterior.get(1).value.toString().equals("int");
        }
        if (!englishChar && !englishInt && anterior.size() > 0 &&
anterior.get(0) != null) {
            englishChar = anterior.get(0).value.toString().equals("char");
            englishInt = anterior.get(0).value.toString().equals("int");
        }
        String err = message + "No se esperaba este componente\n: "
+cur_token.value+".";
        if (englishChar) {
```

```
            message += "\nSe ha encontrado 'char', puede que quisieras
escribir 'car'?\n";
        } else if (englishInt) {
            message += "\nSe ha encontrado 'int', puede que quisieras escribir
'ent'?\n";
        }
        if (info instanceof ComplexSymbol token) {
            List expected = expected_token_ids();
            String tokens = "";
            for (Object t : expected){
                tokens += ParserSym.terminalNames[(int)t] + ", ";
            }
            if (!tokens.isEmpty()) {
                tokens = "Se esperaba algun lexema de los siguientes tipos: "
+ tokens.substring(0, tokens.length() - 2) + ".\n";
            }
            String loc;
            if (token.xleft.getLine() == token.xright.getLine()) {
                loc = "En la linea " + token.xleft.getLine() + " entre las
columnas " + token.xleft.getColumn() + " y " + token.xright.getColumn();
            } else {
                loc = "Desde la linea " + token.xleft.getLine() + " y columna
" + token.xleft.getColumn() + " hasta la linea " + token.xright.getLine() + "
y columna " + token.xright.getColumn();
            }
            err = message + loc + ". \n" + tokens + "Se ha encontrado '" +
token.value + "' de tipo " + ParserSym.terminalNames[token.sym] + ".\n";
        }
        errores += err;
    }

    @Override
    public void report_fatal_error(String message, Object info) throws
Exception {
        report_error("Error fatal: " + message, info);
        done_parsing();
    }

:}
```

Hay tres partes; `init with`, `scan with`, `parser code` : init with va dentro del constructor, scan with se ejecuta cada vez que se coge un símbolo y el parser

code contiene declaraciones y métodos. El arraylist 'anterior' solo contempla los últimos 3 símbolos. Se inicializa con 3 null para que siempre haya 3 valores. Esta variable existe para poder devolver un mensaje de error personalizado (en el método report_error), en concreto si se ha escrito char en vez de car, e int en vez de ent, que son errores propicios.

- **Justificación de la elección del método de análisis escogido**

Se ha escogido el análisis **sintáctico** ascendente con reducciones y desplazamientos ofrecido por CUP porque estamos muy familiarizados con Java y hemos pensado que tenía una buena cantidad de recursos en el aula digital. En cada regla semántica se crea una instancia de un símbolo, que es un nodo del árbol sintáctico generado, para después de realizar el análisis semántico descendente.

Análisis Semántico:

Se ha recorrido el árbol generado por el análisis sintáctico a partir de la raíz *SymbolScript* y a cada caso se ha comprobado que la coherencia semántica se mantuviera. Hay algunos elementos que tienen prioridad en cuanto al procesamiento que otros, pero todos se procesan en orden dentro de su tipo, por lo que todas las declaraciones globales se procesan en orden pero antes que los métodos y después de las tuplas independientemente del orden. Además primero se procesan las declaraciones de las tuplas y los métodos que sus cuerpos, por si en el cuerpo se declaran del tipo tupla en cuestión o se llaman a métodos escritos después en el código. Esto se hace añadiendo los símbolos a la tabla de símbolos para que, los métodos, puedan llamar entre ellos (A llama a B y B llama a A). El último en procesarse es el Main (inicio), porque nadie lo puede llamar. El resto de código es una contemplación tras otra de posibles errores generados por el usuario, y si es así se le avisa con el mensaje indicado. Además cuando se comprueba que no ha habido ningún error se genera el código intermedio de esa parte.

Como ya se ha dicho en el análisis sintáctico, se realiza el análisis semántico descendentemente, esto es porque nos resulta más sencillo de programar, lo cual resulta en un producto de mayor calidad.

Se hace una comprobación exhaustiva de tipos y de que se haya hecho correctamente todo. Los mensajes de error son muy personalizados y explicativos.

Tabla de Símbolos:

La tabla inacabada está inspirada en la de los apuntes, con unos cambios, los parámetros de las funciones son instancias de la clase Parametro y los miembros de las tuplas de DefinicionMiembro. Hemos pensado que de esta manera era más orientado a objetos y que se podía entender mejor. Las variables de tipo array y la definición de las tuplas también tienen sus propias clase, DescripcionArray y DescripcionDefinicionTupla, hijas de DescripcionSimbolo. Las variables, constantes y variables tipo tupla son instancias de DescripcionSimbolo.

La clase *TablaSimbolos* contiene la propia tabla *td*, el nivel actual *n*, la tabla de ámbitos *ta* y la tabla de expansión *te*. Contiene una clase *Entrada* que representa cada entrada de la tabla de expansión. Están todos los métodos necesarios para poner, quitar y modificar elementos de las tablas, incluidos los relacionados con arrays, tuplas y funciones.

Por tanto la estructura es muy sencilla, contiene:

- Tabla de ámbitos: Donde cada elemento representa el inicio de símbolos en *td* que están en el ámbito concreto.
- Tabla de expansión: Contendrá un conjunto de símbolos que quedan ocultos por declaraciones en el bloque actual al profundizar niveles.
- Tabla de descripción: Conjunto de símbolos tratados visibles.

En cuanto a métodos usados para gestionar la tabla de símbolos encontramos:

- **getNivel()** que nos devuelve el nivel actual de la tabla de símbolos
- **vaciar()** que nos permite vaciar todas las tablas asociadas, como pueden ser tabla de ámbitos, tabla de expansión y tabla de descripción.
- **poner()** será el método que usaremos con tal de incorporar un nuevo símbolo dentro de la tabla.
- **sustituir()** que eliminará la descripción de un símbolo y volverá a insertar dicha identificador junto a su descripción dentro de la tabla.
- **entrarBloque()** aumentará el puntero de la tabla de ámbitos.
- **salirBloque()** recogerá todas las entradas de la tabla de expansión y las incorporará dentro de la descripción. Acto seguido eliminará todas las de expansión que han sido traspasadas.
- **consulta()** devuelve la descripción de un símbolo pasado por parámetro.
- **sePuedeDeclarar()** método con el que se comprueba si un identificador ha sido declarado con anterioridad en el mismo nivel, o si coincide con una tupla o método, en caso de que haya un identificador igual a un nivel superior, se puede declarar.

Optimizaciones:

A la hora de llevar a cabo el proyecto se han planteado una serie de optimizaciones, estas cambian un poco la estructura del código de 3 direcciones y por ende afectan sobre el fichero con código en lenguaje máquina. Cabe aclarar que las salidas de los ficheros con y sin optimizar es la misma. Las optimizaciones realizadas son las siguientes:

- Ramificación sobre ramificación: Si se llegan a generar saltos que llevan a otros saltos lo que se hace es cambiar el primer salto por el salto que llevaba la etiqueta a la que se hace referencia.

- Ramificaciones adyacentes: Se niegan las condiciones con tal de poder hacer más eficientes las comparaciones, reduciendo saltos innecesarios. Esta optimización puede dejar etiquetas que no son utilizadas. La siguiente optimización se encargará de gestionar eso.

- Eliminaciones de skips no usados: Con tal de reducir instrucciones que no son usadas, se procederá a eliminar cada uno de las etiquetas a las cuales no se hagan llamadas. Esta optimización es realmente útil junta a las dos mencionadas.

- Asignaciones diferidas: Si tenemos variables que solo se usan una única vez sobre otra variable, se le asigna el valor directamente y se elimina la variable.

Restricciones:

1. No se pueden hacer asignaciones complejas sobre arrays (+:, -:, …).
2. Los incrementos/decrementos solo se pueden hacer sobre variables primitivas (arr[0]++ y tupl.miem++ no están permitidos).
3. No se pueden realizar varias operaciones de incremento/decremento seguidas (a++-- y ++a-- no están permitidos incluso con paréntesis).
4. No pueden haber dos métodos con el mismo nombre (no se permite sobrecarga).
5. Las tuplas solo pueden ser declaradas fuera de los métodos y de otras tuplas.
6. Los métodos se han de declarar fuera de cualquier otro método.
7. Las dimensiones de los arrays tienen que ser valores enteros literales y sin operaciones (ent[3][7]arr está permitido pero ent[2+'3'][constante]arr no).
8. No pueden haber arrays de tuplas.
9. No pueden haber arrays ni tuplas dentro de tuplas.
10. No puede haber llamadas recursivas
11. No se pueden imprimir vocales con tildes, ñ, comillas simples o comillas dobles
12. Cuando se imprime un número, se imprime el carácter al que corresponde el valor numérico que tenía, pero primero hay que pasarlo a *car* con casting y luego a *car[]*
13. No se pueden realizar asignaciones entre array ni entre tuplas (arr1[], arr2[], arr1: arr2 no está permitido)

Notas:

Las variables tupla se inicializan al declararse. Los arrays cuando se declaran con las dimensiones también son inicializados (arr[1][2]). Cuando se declaran sin las dimensiones aún no han sido inicializados (arr[][]).

El código ensamblador se ha realizado para el emulador EASy68K, el cual es gratuito y se puede descargar desde aquí: http://www.easy68k.com/

Casos de prueba:

Los casos de prueba del proyecto se encuentran en 2 carpetas diferentes, por una parte para los scripts con correcto funcionamiento los podemos encontrar en */Dartam/scriptsCorrectos*, en cambio aquellos que contienen algún error, ya sea léxico, sintáctico o semántico se encuentran en */Dartam/scriptsErroneos*.

Los scripts vistos a continuación son solo un pequeño conjunto del que se puede encontrar dentro de las carpetas con ejemplos suministrados.

Scripts Erróneos:

1) errLex1.dtm

```
f vacio inicio(){
	ent gradiosCelsiusº: 5; // variable tipo ent con nombre
inválido
}
```

Este primer ejercicio, da un error léxico debido a que el símbolo "º" no está dentro del alfabeto del analizador léxico.

2) errSem1.dtm

```
f vacio inicio(){
	inmut ent a, b:20;
	b: 10; // asignando a constante ya asignada un valor
	ent c: a; // asignando a variable una constante sin valor
	car d: b; // asignando a variable una constante de diferente
tipo
}
```

Vemos 3 errores diferentes en este caso:

- En la línea 3 se asigna un valor a una constante ya asignada
- En la línea 4 se asigna a la variable "c" la constante "a" sin ningún valor asociado.
- En la línea 5 se hacen asignaciones con variables de diferentes tipo, en este caso "d" de tipo car y "b" de tipo ent.

3) errSin1.dtm

```
f vacio main(){
	// no se ha encontrado metodo inicial (inicio)
}
```

En este caso el error originado viene del hecho de que todos los ficheros que vayamos a compilar deben tener un método *f vacio inicio(){... }* si este no se encontrará dará un error.

Scripts Correctos:

1) persona.dtm

```
inmut ent TURNOS;

f vacio inicio(){
	TURNOS: 3;
```

```
	car[] str: "";
	tupla persona p1, p2;
	nuevaPersona(p1, 1, 'a');
	nuevaPersona(p2, 2, 'm');
	show("Bienvenido a la simulación");
	show("Dos personas pensarán hasta que tengan hambre, momento
en el que comerán");
	show("");
	loop ent i:0; i < TURNOS; i++ {
		car[] str: "La persona";
		p1.hambre: p1.hambre + 1;
		p2.hambre: p2.hambre + 1;
		si ¬((p1.hambre > p1.maxHambre) | (p2.hambre >
p2.maxHambre)) {
			imprimirTexto("La persona", (car[])p1.inicial,
"piensa");
			imprimirTexto("La persona", (car[])p2.inicial,
"piensa");
			continuar;
		}
		si p1.hambre > p1.maxHambre { // p1 tiene prioridad
			str: (car[])p1.inicial;
			p1.hambre: 0;
		} no {
			str: (car[])p2.inicial;
			p2.hambre: 0;
		}
		imprimirTexto("La persona", str, "come");
	}
	show(str);
	show("Escribe lo que te ha parecido");
	show("El resultado se guardará en opinion.txt");
	scan(str);
	into("opinion.txt", str);
	show("Fin");
}

f vacio nuevaPersona(tupla persona p, ent maxHambre, car inicial) {
	si maxHambre > TURNOS {
		show("La persona con inicial");
		show((car[])inicial);
		show("Nunca pasará hambre");
	}
	p.maxHambre: maxHambre;
	p.inicial: inicial;
}
```

```
	tupla persona {
		inmut ent maxHambre;
		car inicial;
		ent hambre: 0;
	}

	f vacio imprimirTexto(car[] s1, car[] s2, car[] s3) {
		show(s1);
		show(s2);
		show(s3);
		show("");
	}

}
```

En este ejemplo se incluyen declaraciones de cadenas de caracteres; definición, declaración y asignación de tuplas, cómo acceder a sus valores, asignarlos; bucles del estilo for (`loop ent i:0; i < TURNOS; i++`); operadores lógicos como not (¬) y or (|).

La tupla persona está definida como un conjunto de valores: valor entero constante `maxHambre`, un carácter `inicial` y un entero `hambre` inicializado a 0.

2) prod.dtm

```
	f vacio inicio(){
		ent a : product(2,3);
		si a = 6 {
			show("2 * 3 = 6");
			show("6 esta almacenado en a");
		} no {
			show("ERROR: 2*3 no ha dado 6");
		}
	}

	f ent product(ent x, ent y){
		ent z : 0;
		loop x /= 0 {
			si (x \ 2) = 1 {
				z +: y;
			}
			y *: 2;
			x /: 2;
		}
		pop z;
	}
```

En este código se puede ver la correcta utilización de un bucle del estilo while, asignaciones, operadores con asignación y llamada a funciones.
En el main (`inicio()`), se asigna al entero `a` el valor resultante de la ejecución de la función `product(ent x, ent y)`, que devuelve el producto de dos enteros utilizando el método de sumas sucesivas. Posteriormente se hace una comprobación por texto y simplemente imprime `"6 esta almacenado en a"` si la operación ha salido bien, en caso contrario imprime el error.

3) io.dtm

```
f vacio inicio(){
	show("Escribe el fichero a leer:");
	car[81] file, content;
	scan(file);
	from(file, content);
	show("El fichero contiene:");
	show(content);
	show("Ahora escribe en el fichero:");
	scan(content);
	into(file, content);
	show("Ahora el fichero contiene:");
	from(file, content);
	show(content);
}
```

En este ejemplo se usan operaciones de entrada/salida. La entrada por teclado con `scan(file)`, la salida por pantalla con `show()`, la lectura de fichero con `from(file, content)`, y la escritura con `into(file, content)`

Vídeo explicativo

En YouTube: https://youtu.be/DTrWT9vuvX0