EIF400-II-2025 Paradigmas de Programación

Sprint Final. Proyecto Expresso

- Version: alpha
- Fecha: 7/10/2025
- `loriacarlos@gmail.com`

RESUMEN DE FEATURES ESPERADOS

1. Se soporta todo lo pedido en sprints anteriores.Incluyendo el comando `expressor` de consola del spec original.
2. Se soportan tipos `Object` (`any` en `Expresso`), flotantes (`float`) booleanos (`boolean`) e hileras (estilo `Java` con `"`). (`string`). Para tipo de usuario, se soportan declaraciones `data` que crean tipos de usuario, según se explica adelante.

```
let msg = "Hello \"World\"\n"
print(msg)
/*
 Salida
 Hello "World" seguida de cambio de línea
*/
```

3. Está disponible la declaración `fun` para lambdas como métodos. Es una declaración estatuto como un `let` pero permite una definición recursiva. Ejemplo:

```
fun fact(n:int):int = n ? n * fact(n - 1) : 1
```

Note que como antes se permite `int` en la condición de un booleano por excepción.

4. Hay tipos explícitos. Una expresión se anota con su tipo usando el operador `:` . Si no se declara se asume `any` en esta versión; o en casos como un `let` se infiere del tipo del lado derecho. Por ejemplo:

```
let PI:float = 3.14
let x = 666 // lo mismo que let x:int = 666
fun cat(s:string t:string):string = s + t
let my_pow:((float, int) -> float) =
```

```
            (x, n) -> x ** n
let another_my_pow = (x:float, n:float)
                       -> x ** n
```

Más adelante se describen más detalles sobre los tipos y su lógica.

5. Hay declaraciones `data` para datatypes, como se mencionó ya en el spec de arranque. Son sintácticamente estatutos como el `let`. Recuerde que se transpilan como `sealed` interfaces y `permits` con `record` para las implementaciones (ver spec original del proyecto). Ejemplos:

```
// Números naturales

data nat = {
  Zero,
  S(n:nat)
}
// Listas tipo Scheme/Prolog
data list = {
  Nil,
  Cons(car:any, cdr:list)
}
 // Simulando un enum
data gender = {
   Male, Female
}
```

Lo que va dentro de los `{...}` separados por comas se llaman **constructores**. La sintaxis de una declaración `data` es así (`flatType` se define más adelante, son cosas como `int`, `int -> int`, etc.):

```
data : 'data' id '=' {' constructores '}';
constructores : constructor
                ( ',' ,constructor)*
constructor : id arguments?;
arguments : '(' argument
              (',' argument)* ')'
argument : (id ':')? flatType;
```

6. Hay un operador `^` (es prefijo y no es associativo) que es para instanciar constructores de data types. Se transpila como un `new`. Con este hay una nueva expresión (note que `^` tiene una sintaxis y tratamiento especial). Ejemplo:

```
let my_list = ^Cons(666, ^Nil)
```

Lo anterior transpila a

```
var my_list = new Cons(666, new Nil());
```

Otro ejemplo: Acá se usa `match`:

```
let first = a -> match a  with
      Nil -> none
        | Cons(f, _) -> f
print(first(my_list)) // Imprime 666
print(first(^Nil)) // imprime none
```

La sintaxis es así (tienen que adaptar sus gramáticas según cada caso).

```
expr : ...
       | '^' construtor_expr
           ...
;
construtor_expr : id ( '(' params ')' )?;
params : expr, (',' expr)* ;
```

7. Hay una expresión `match` que es una versión simplificada del `switch` de pattern-matching de `Java 21`+. Sintaxis:

```
expr : ...
       | 'match' expr `with` rule+
           ...
;
rule : pattern guard? -> expr
;
guard : expr
;
pattern : data_pattern
        | native_pattern
;
data_pattern ->
      id ('(' data_pattern+ ')')?
;
native_pattern ->
       'none'
   | string_constant
   | numeric_constant
   | boolean_constant
     // Bug in jdk 24
```

```
;
```

8. El `print` puede ser también expresión. Su valor es un objeto especial `none` que es el único objeto de tipo `void`. Este valor es una constante como `true` y `false`. Por ende es una expresión también. Se puede transpilar a `null`.

```
let x = none // es válida. x es de tipo void
let y = print("Hello World") // válida. y:void
```

9. Los tipos tienen su propio lenguaje y por ende gramática particular.

```
// Grammar of types:
type : flatType | tuple | '(' type ')'
;

flatType : atomic | arrow
;

atomic -> 'any'       // -> Object
    | 'void'    // -> el mismo
    | 'int'     // -> Integer
    | 'float'   // -> Double
    | 'string'  // -> String
    | usergiven // -> el mismo
;
tuple : '(' flatType (',' flatType)* ')' }
;        // No se puede anidar tuples
arrow : type '->' flatType;
// '->' Asocia derecha.
// No es posible retornar un tuple.
usergiven :
// un nombre creado con una declaración data.
```

10. Hay operadores relacionales (mínimo `<`, `<=`, `==`, `!=` y booleanos (`&&`, `||`, `!`) y constantes `true` y `false`. Se comportan como en Java.

11. Se permite como antes que ternario reciba en su condición algo de tipo `int`. Y se hace la transformación como antes. En otro caso debe ser de tipo boolean.

12. Hay una expresión para `casting`, se usa el operador binario `:` No es asociativa. Tiene más precedencia que `**`.

```
expr :
      ...
      | expr ':' id
```

```
      ...
;
```

Ejemplo:

```
let x = 2.5
print(3**2*10+x:int)
// Parsea como: (3 ** 2) * 10 + (x:int)
```

Para efectos de transpilación: `expression:type` equivale a un casting: `((java_type)expression)` donde `java_type` es el tipo resultante de la transpilación del tipo de `expresso` `type` al tipo de `Java` correspondiente.

Por ejemplo, `int -> int` transpila a `UnaryOperator<Integer>`.

Principios del Analizador Semántico (Typer)

> Nota: El requerimiento del typer está por definirse si se mantiene como obligatorio o es opcional. Sin embargo la declaración de tipos si es obligatoria de manejarse así como su transpilación a Java.

El typer en esta versión sería solo un validador de tipos, no hace inferencia de tipos, aún. Se dice que es un "verificador" de tipos. Valida también posible nombres no definidos pero solo de manera restringida. Es decir, en esta versión esa funcionalidad estará incompleta pues no se procesan los `import` ni tendremos nombres calificados. Solo se validan nombres que el mismo archivo declare.

Función del Typer a grandes rasgos

- El typer va a visitar el `AST` de un programa y va a tratar de descubrir problemas de tipos o nombres no definidos.
- El typer concoce los tipos de operadores primitivos y de expresiones. Por ejemplo, sabe que `true` es de tipo `boolean` o que `"hello"` es de tipo `string`.
- Los tipos son objetos que tienen su propio `AST` y una lógica proposicional que opera sobre sobre ellos (en este caso por ser un verificador).
- El objetivo del typer es **asignarle un único tipo a cada nodo del `AST` del programa**.

Por ejemplo, suponga que en su visita (en un contexto dado) el typer ya probó que `x` y `y` son de tipo `string`. Entonces si ve el `AST` de `x + y` probabría que ese nodo es de tipo `string` (`+` es concatenación de hileras). Si luego en ese mismo contexto visita la lambda `(x, y) -> x + y` puede asignarle tipo `(string, string) -> string`. Si se trata de usar esa lambda con tipos no compatibles el typer reporta el error.

Otro ejemplo: al nodo asociado con `x * 666` le tocaría un tipo `int` siempre que se haya probado que `x` recibió un tipo `int` en el contexto. Y 666 es una contante entera, eso asume el typer.

Se basa en reglas simples: Por ejemplo, suponga que el typer sabe que `+` es un operador binario de dos `int` y que retorna un `int`, entonces una declaración como la siguiente no pasa el typer:

```
fun add(x:string):int = x + 1
```

El typer visita el fun y recoge la afirmación que add tiene tipo `string -> int`. Y registra que en ese contexto `x:string`. Visita el lado derecho y ve la suma `+`. Esa suma tiene tipo `(int, int) -> int`. Eso lo sabe el typer de en sus estrcuturas de datos, de manera fija.

Dado esto el typer se da cuenta que `x` debería ser `int`, pero tiene en el contexto de la función tiene tipo `string`, donde el type sabe que `int` y `string` no son *compatibles*. Por ende rechaza la declaración y termina su trabajo (lanazando una excepción de tipos, es decir, un error de semántica estática).

Note que sería un typer poco *tolerante* para esta versión: por cada error se va a detener. Esto en la práctica sería **muy inusable**. Hay técnicas mejores que buscan capturar la mayor catidad de errores de una vez. En nuestra versión no aplicaremos esas estrategias.

Detalle Más Específicos

Detalles específicos se explicarán en clase el 9/10 y subsiguientes.

Forma y Criterios de Evluación

La evaluación y los criterios serán totalmente similares a los de los sprint previos. Habrán casos de prueba y revisión y defensa de los resultados. La funcionalidad vale 65%, la defensa y revisión del código 35%. Los valores de los casos de prueba y los mismos serán comunicados

con antelación. Habrá como a antes una guía de evaluación para registrar los datos de la misma.

Tipos de Casos de Prueba

Use los ejemplos de este Spec para construir pequeños casos de prueba. Trabaje incrementalmente de casos simples a más complejos.

Fecha y forma de entrega

Mismo procedimiento de antes: se sube en formato zip al **mismo drive** que antes. Un proyecto Java bien organizado mismas condiciones de sprints previos que permitan su construcción, prueba y revisión de manera efectiva. El nombre del zip `EIF400-II-2025_Expresso_Final_NNNN_GG_HH.zip` (`NNNN` nombre del coordinador, `GG_HH` código de grupo asignado, como antes). Contiene una única carpeta `expresso` que tiene el proyecto. El `README` permite lo necesario sin retrasos ejecutar la tarea de revisión.

Fecha Subida al drive: Domingo 9/11 12md.

Fecha de demo y defensa: Lunes 10/11 en los horarios de matrícula.