Extensión del lenguaje LIS (LIS-E)

Información General del Proyecto

Propósito: Definición formal e implementación de una extensión del Lenguaje Imperativo Simple (LIS).
Lenguaje de implementación: Haskell, utilizado como lenguaje subyacente de especificación y ejecución.
Asignatura: Teoría de Lenguajes – AUS, UNR-IPS (Rosario, Argentina).

Instrucciones rápidas

---

1. Clonar el repositorio:
   `git clone https://github.com/Andrenon/aus-2025-teoria-de-lenguajes`
   `cd aus-2025-teoria-de-lenguajes`
2. LIS-E es un lenguaje interpretado cuya ejecución requiere como argumento un archivo `<nombre>.lis`. (Opcional compilar el intérprete Haskell con `ghc`)
3. Para pruebas:
   `ghci Main.hs`
4. Ejecutar desde ghci:
   `run "test/test01.lis"`
   `run "test/test02.lis"`
   `...`

Descripción

En este trabajo se presenta una extensión del lenguaje imperativo simple LIS, denominada LIS-E, cuyo objetivo es incorporar:

- Soporte nativo para strings
- Operadores de conversión entre enteros y strings
- Operador de concatenación para strings (++)
- Operador de flujo |> definido en dos niveles sintácticos:
  - A nivel de expresiones, como azúcar sintáctico para la composición funcional.
  - A nivel de comandos, como un constructor sintáctico que modela la aplicación de acciones con efectos sobre un flujo.
- Comandos con *efectos* observables sobre valores intermedios
- Capturar *errores* en tiempo de ejecución (semántica)

Extensión del *dominio* de valores:

$$Val ::= Z \cup String$$

El programa completo es un comando. Semánticamente:

$$(c, \sigma) \Downarrow (\sigma\prime, out, err)$$

donde:

- c: es el comando a ejecutar
- σ, σ′: estado inicial y final, respectivamente
- out: representa los efectos observables acumulados
- err: indica la posible ocurrencia de un error dinámico
  La evaluación de un programa produce un nuevo estado junto con su comportamiento observable y el eventual reporte de error.

Se mantiene que:

> Todo programa válido en LIS es válido en LIS-E y tiene la misma semántica.
> $(LIS \subseteq LIS - E)$

Diseño del lenguaje LIS-E

Nivel 1 — Sintaxis

- `AST.hs` → Sintaxis abstracta
- `Parser.hs` → Sintaxis concreta
  Esto define el **lenguaje**, no su ejecución.

Nivel 2 — Semántica

- `Eval2.hs` → Semántica operacional big-step (evaluador monádico)
  Especificación ejecutable del lenguaje: `⟨c,σ⟩⇓(σ′,out,err)`.
  El evaluador se construye mediante la composición de mónadas:
  - `ExceptT` → modela manejo de errores dinámicos
  - `StateT` → modela el estado (store)
  - `Writer` → modela los efectos observables (log)

Nivel 3 — IO

- `Main.hs` → Interfaz interactiva
  Conecta: `archivo → parser → evaluador → print resultado`

Sintaxis abstracta

```
Value ::= IntVal Int | StrVal String

intexp ::= ...
         | len strexp
         | toInt strexp

strexp ::= STRING | svar | toStr intexp
         | strexp ++ strexp
         | strexp |> strfilter

strfilter ::= upper | lower | reverse | trim

comm ::= ...
       | svar ::= strexp
       | step strexp
       | comm |> pipe_action

pipe_action ::= strfilter | print | sleep intexp
```

Sintaxis concreta

```
intexp ::= ...
         | 'len(' strexp ')'
         | 'toInt('strexp ')'

strexp ::= strexp_no_pipe | strexp '|>' strfilter
strexp_no_pipe ::= strexp_value | strexp_no_pipe '++' strexp_value
strexp_value ::= STRING | svar | 'toStr(' intexp ')'

strfilter ::= 'upper' | 'lower' | 'reverse' | 'trim'

comm ::= ...
       | svar '::=' strexp
       | 'step(' strexp ')'
       | comm '|>' pipe_action

pipe_action ::= strfilter | comm_effect
comm_effect ::= 'print' | 'sleep(' intexp ')'
```

Semántica formal

Se respeta:

- Expresiones → big-step (⇓)
- Comandos → small-step (⇝)

Se introduce un nuevo comando intermedio:
`pipe(w,k)`

Este constructor:

- no es visible al usuario
- es interno de la semántica
- representa pipeline en ejecución (w = valor actual, k = número de pasos ejecutados)

Expresiones enteras nuevas

E-Len

$$\frac{\langle s, \sigma\rangle \Downarrow_{strexp} w}{\langle len(s), \sigma\rangle \Downarrow_{intexp} | \ w \ |}$$

E-ToInt

$$\frac{\langle s, \sigma\rangle \Downarrow_{strexp} w \quad parse(w) = n}{\langle toInt(s), \sigma\rangle \Downarrow_{intexp} n}$$

Expresiones string

E-String

$$\frac{}{\langle "c", \sigma\rangle \Downarrow_{strexp} \ "c"}$$

E-SVar

$$\frac{\sigma(s) = w}{\langle s, \sigma\rangle \Downarrow_{strexp} w}$$

E-ToStr

$$\frac{\langle e, \sigma\rangle \Downarrow_{intexp} n}{\langle toStr(e), \sigma\rangle \Downarrow_{strexp} str(n)}$$

E-Concat

$$\frac{\langle s1, \sigma\rangle \Downarrow_{strexp} w1 \quad \langle s2, \sigma\rangle \Downarrow_{strexp} w2}{\langle s1 ++ s2, \sigma\rangle \Downarrow_{strexp} w1 \cdot w2}$$

Pipe a nivel de expresiones

Azúcar sintáctica: `s|>f ≡ f(s)`

E-Pipe

$$\frac{\langle s, \sigma\rangle \Downarrow w}{\langle s \mid> f, \sigma\rangle \Downarrow_{strexp} Ff(w)}$$

Donde:

filtro	función matemática
upper	uppercase(w)
lower	lowercase(w)
reverse	reverse(w)
trim	removeSpaces(w)

Comandos

C-SAssign

$$\frac{\langle s, \sigma\rangle \Downarrow_{strexp} w}{\langle x ::= s, \sigma\rangle \rightsquigarrow \langle skip, \sigma[x \mapsto w]\rangle}$$

C-Step

$$\frac{\langle s, \sigma\rangle \Downarrow_{strexp} w}{\langle step(s), \sigma\rangle \rightsquigarrow \langle pipe(w, 0), \sigma\rangle}$$

Inicia un flujo observable.

Pipe a nivel de comandos

Constructor semántico: `c |> a` introduce una acción con efectos sobre un flujo activo.

C-Pipe-Filter

$$\frac{}{\langle pipe(w, k) \mid> f, \sigma\rangle \rightsquigarrow \langle pipe(Ff(w), k+1), \sigma\rangle}$$

C-Print

$$\frac{}{\langle pipe(w, k) \mid> print, \sigma\rangle \rightsquigarrow \langle pipe(w, k+1), \sigma\rangle}$$

C-Sleep

$$\frac{\langle e, \sigma\rangle \Downarrow_{intexp} n}{\langle pipe(w, k) \mid> sleep(e), \sigma\rangle \rightsquigarrow \langle pipe(w, k+1), \sigma\rangle}$$

Explicación coloquial de las extensiones

`len(s)`
Devuelve la longitud del string evaluado.

`toInt(s)`
Convierte un string numérico a entero.
Produce error si el string no representa un entero válido.

`toStr(e)`
Convierte un entero en string.

`++`
Concatena dos strings.

`step(s)`
Inicia un pipeline.
Permite observar transformaciones intermedias.

`|>`
Aplica una transformación o efecto al flujo actual.

`upper / lower / reverse / trim`
Transformaciones puras sobre strings.

`print`
Imprime el valor del pipeline. Operador de efecto lateral no terminal.

`sleep(n)`
Suspende la ejecución n segundos sin modificar el valor.

Conclusión

LIS-E constituye una extensión conservativa del Lenguaje Imperativo Simple (LIS), incorporando soporte nativo para strings y un mecanismo explícito de flujo mediante pipelines.

La extensión no altera la semántica de los programas válidos en LIS, preservando así la compatibilidad hacia atrás (LIS $\subseteq$ LIS-E). No obstante, amplía el dominio semántico del lenguaje al incorporar efectos observables y manejo explícito de errores como parte del resultado de evaluación.

El diseño mantiene una separación clara entre sintaxis abstracta, sintaxis concreta, semántica operacional e implementación del evaluador, asegurando coherencia formal y modularidad en su construcción.