Parseo y Generación de Código

Intérpretes

Licenciatura en Informática con Orientación en Desarrollo de Software
Universidad Nacional de Quilmes

Intérpretes

Introducción

Los lenguajes de programación tienen dos aspectos: **sintaxis** y **semántica**.

- **Sintaxis:** ¿qué forma tienen los programas válidos?
- **Semántica:** dado un programa válido, ¿qué significa?

Introducción

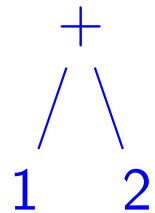
La palabra "lenguaje" tiene dos acepciones:

1. **Acepción sintáctica:** un lenguaje en este sentido es un conjunto de cadenas en un alfabeto. Esta es la definición que nos interesaba al momento de hacer análisis sintáctico.

   Por ejemplo, bajo esta acepción, la cadena "1 + 2" está en el lenguaje de expresiones aritméticas, y la cadena "2 *" no está en dicho lenguaje.

2. **Acepción semántica:** un lenguaje en este sentido está dado por un conjunto de programas (que pueden ser estructuras de distintos tipos; por ejemplo, árboles) que vienen acompañada de una noción de "significado" de un programa.

   Por ejemplo, bajo esta acepción, en algún lenguaje, un programa podría ser el siguiente árbol, y su significado podría ser el número 3.



Introducción

En la primera clase escribíamos $\mathcal{L}(P, X)$ para representar el significado del programa P en el lenguaje $\mathcal{L}$ cuando se le pasa la lista de parámetros X.

Un **intérprete** para un lenguaje $\mathcal{L}_{\texttt{fuente}}$ es un programa I escrito en un lenguaje $\mathcal{L}_{\texttt{implementación}}$, que cumple la siguiente igualdad:

$$\mathcal{L}_{\texttt{fuente}}(P, X) = \mathcal{L}_{\texttt{implementación}}(I, [P]{+}{+}X)$$

para cualquier lista de parámetros X.

Es decir: el significado del programa P en el lenguaje $\mathcal{L}_{\texttt{fuente}}$ es el mismo que el significado del intérprete I en el lenguaje $\mathcal{L}_{\texttt{implementación}}$ cuando se le pasa el programa P como parámetro.

Introducción

- Interpretar un lenguaje $\mathcal{L}_{\texttt{fuente}}$ en un lenguaje $\mathcal{L}_{\texttt{implementación}}$ puede ser difícil. Por ejemplo:
  - Podría ser que $\mathcal{L}_{\texttt{fuente}}$ tenga números enteros arbitrariamente grandes pero $\mathcal{L}_{\texttt{implementación}}$ no.
  - Podría ser que $\mathcal{L}_{\texttt{fuente}}$ tenga funciones de "alto orden" y *pattern matching* pero $\mathcal{L}_{\texttt{implementación}}$ no.
  - Podría ser que $\mathcal{L}_{\texttt{fuente}}$ permita lanzar excepciones pero $\mathcal{L}_{\texttt{implementación}}$ no.
- Vamos a suponer que el problema de análisis sintáctico ya está resuelto, y vamos a hacer intérpretes manipulando directamente el AST de los programas.

Intérprete para un lenguaje de expresiones

Empecemos por un lenguaje de expresiones minimal:

```
data Expr = ExprConstNum Int          -- n
          | ExprAdd Expr Expr         -- e1 + e2
```

se puede interpretar fácilmente:

```
eval01A :: Expr -> Int
```

Ejercicio. Definir el intérprete.
Notaremos $[\![e]\!]$ en lugar de `(eval01A` e`)`.

Intérprete para un lenguaje de expresiones

Un concepto central en los intérpretes de lenguajes de programación es el de **entorno** (*environment*). Un entorno es un diccionario que asocia identificadores a valores.

Supondremos que contamos con un tipo `Id` para los identificadores (nombres de variables), un tipo `(Env a)` y funciones:

- `emptyEnv :: Env a`
- `lookupEnv :: Env a -> Id -> a`
- `extendEnv :: Env a -> Id -> a -> Env a`

Notaremos E a los entornos, y escribiremos las operaciones así:

$\varnothing$	para	`emptyEnv`
$E(x)$	para	`lookupEnv` E x
$E[x := v]$	para	`extendEnv` E x v

Intérprete para un lenguaje de expresiones

Extendamos el intérprete con:

- booleanos
- variables (`x`)
- declaraciones locales (`let x = e1 in e2`)

```
data Expr = ExprConstNum Int          -- n
          | ExprConstBool Bool        -- b
          | ExprAdd Expr Expr         -- e1 + e2
          | ExprVar Id                -- x
          | ExprLet Id Expr Expr      -- let x = e1 in e2
```

Contamos ahora con el tipo de los posibles valores:

```
data Val = VN Int
         | VB Bool
```

El tipo de la función `eval` se modifica para incorporar un entorno:

```
eval02Env :: Expr -> Env Val -> Val
```

Ejercicio. Definir el intérprete.
Notaremos $[\![e]\!]E$ en lugar de (`eval02Env` e E).

Ejercicio. Evaluar `(let x = 1 in (let x = x in x + x))`.

Intérprete para un lenguaje de expresiones

- En nuestro primer lenguaje, el significado de un programa era un número.
- ¿Cuál es el significado de un programa en nuestro segundo lenguaje?
- Para responder esta pregunta puede ser conveniente escribir todos los paréntesis en el tipo de la función `eval02Env`:

```
eval02Env :: Expr -> (Env Val -> Val)
```

- Si e `::` `Expr`,
  entonces $[\![e]\!]$ `::` `Env Val -> Val`.

Características imperativas

Intérprete *store-passing* para un lenguaje imperativo

Extendamos el lenguaje con las siguientes características imperativas:

- Asignaciones:
  x `:=` e supondremos que tiene un efecto y su valor es 0.
- Componer dos programas en secuencia:
  e_1 `;` e_2 ejecuta el efecto de e_1 y descarta su resultado.

```
data Expr = ExprConstNum Int          -- n
          | ExprConstBool Bool        -- b
          | ExprAdd Expr Expr         -- e1 + e2
          | ExprVar Id                -- x
          | ExprLet Id Expr Expr      -- let x = e1 in e2
          | ExprSeq Expr Expr         -- e1; e2
          | ExprAssign Id Expr        -- x := e
```

- La función `eval02Env :: Expr -> (Env Int -> Int)` que teníamos no alcanza para definir el intérprete de manera **composicional**:

$$[\![(x := e_1); e_2]\!]E = ???$$

Intérprete *store-passing* para un lenguaje imperativo

- En un lenguaje con características imperativas (como Java, Python, Ruby, JavaScript, etc.) las variables no están ligadas a **valores** sino a **direcciones de memoria**.
- Cada dirección de memoria contiene un valor.

Intérprete *store-passing* para un lenguaje imperativo

En un intérprete para un lenguaje imperativo el concepto central es el del **memoria**. La memoria asocia direcciones a valores.

Supondremos que contamos con un tipo `Addr` para las direcciones de memoria, un tipo `(Memory a)` y funciones:

- `emptyMemory :: Memory a`
  devuelve una dirección de memoria sin usar.
- `allocate :: Memory a -> Addr`
  devuelve una dirección de memoria sin usar.
- `store :: Memory a -> Addr -> a -> Memory a`
  almacena el valor dado en la posición indicada.
- `dereference :: Memory a -> Addr -> a`
  devuelve el valor almacenado en la posición indicada.

Notaremos M a las memorias, y escribiremos a las operaciones así:

$\varnothing$	para	`emptyMemory`
$M(a)$	para	`dereference` M a
$M[a := v]$	para	`store` M a v

Intérprete *store-passing* para un lenguaje imperativo

Podemos definir ahora una nueva función:

```
eval03Mem :: Expr -> Env Addr -> Memory Val
                              -> (Val, Memory Val)
```

Ejercicio. Definir el intérprete.
Notaremos $[\![e]\!]\, E\, M$ en lugar de (`eval03Mem` e E M).

¿Cuál es el significado de un programa en nuestro tercer lenguaje?

```
data Expr = ExprConstNum Int          -- n
          | ExprConstBool Bool        -- b
          | ExprAdd Expr Expr         -- e1 + e2
          | ExprVar Id                -- x
          | ExprLet Id Expr Expr      -- let x = e1 in e2
          | ExprSeq Expr Expr         -- e1; e2
          | ExprAssign Id Expr        -- x := e
```

Intérprete *store-passing* para un lenguaje imperativo

Ejercicio. Extender el intérprete a una función `eval04MemControl` que incorpore las estructuras de control `if` y `while`.

```
data Expr = ...
   | ExprLtNum Expr Expr     -- e1 < e2
   | ExprIf Expr Expr Expr   -- if e1 then e2 else e3
   | ExprWhile Expr Expr     -- while e1 do e2
```

Intérprete *continuation-passing* para un lenguaje imperativo

En lugar de devolver los resultados recursivamente, se puede trabajar con intérpretes que pasen los resultados a una **continuación**. Una continuación es una función que recibe un valor y un estado de la memoria y hace algo con ellos, devolviendo un resultado final.

El tipo de las continuaciones está dado por

```
type Cont = (Val, Memory Val) -> Result
```

Usando continuaciones podemos definir una nueva función

```
eval05Cont :: Expr -> Env Addr -> Memory Val
                   -> Cont -> Result
```

Intérprete *continuation-passing* para un lenguaje imperativo

Ejercicio. Definir un intérprete *continuation-passing* para el lenguaje de recién:

```
eval05Cont :: Expr -> Env Addr -> Memory Val
                   -> Cont -> Result
eval05Cont = ?

data Expr =
     ExprConstNum Int          -- n
   | ExprConstBool Bool        -- b
   | ExprAdd Expr Expr         -- e1 + e2
   | ExprVar Id                -- x
   | ExprLet Id Expr Expr      -- let x = e1 in e2
   | ExprSeq Expr Expr         -- e1; e2
   | ExprAssign Id Expr        -- x := e
   | ExprIf Expr Expr Expr     -- if e1 then e2 else e3
   | ExprWhile Expr Expr       -- while e1 do e2

type Cont = (Val, Memory Val) -> Result
```

Intérprete *continuation-passing* para un lenguaje imperativo

Comentario. El pasaje de continuaciones puede parecer una técnica innecesariamente rebuscada, pero tiene muchas aplicaciones:

- Se puede utilizar para implementar muchas estructuras de control:
  - Excepciones.
  - *Conditions* (p.ej. de Common-Lisp).
  - *Generators* (p.ej. de Python, Icon).
  - *Backtracking* (p.ej. Prolog).
  - Corrutinas (p.ej. Python).
  - `call/cc` (p.ej. Scheme).
- Sirve para forzar un orden de evaluación (p.ej. para para dar evaluar un lenguaje call-by-value cuando el lenguaje de implementación es call-by-name).
- La técnica de pasaje de continuaciones se usa como lenguaje intermedio para compilar lenguajes funcionales.

Intérprete *continuation-passing* para un lenguaje imperativo

Una característica importante de la técnica radica es que el intérprete tiene control explícito del contexto de ejecución del programa.

Por ejemplo si el tipo de los resultados finales `Result` es el mismo que el tipo de los valores `Val`, se puede extender el lenguaje con una construcción que aborta la ejecución y devuelve el número dado:

```
data Expr = ...
          | ExprAbort Int     -- abort(n)
```

Se evalúa de la siguiente manera:

$$[\![\texttt{abort(n)}]\!]\, E\, M\, k = \texttt{VN}\ n$$

Características funcionales

Intérpretes para lenguajes funcionales

Los lenguajes funcionales están casi unánimemente basados en el cálculo-λ. El cálculo-λ es un lenguaje funcional minimal que tiene solamente tres construcciones:

```
data Expr =
         ExprVar Id                -- x
       | ExprLam Id Expr           -- λx.e
       | ExprApp Expr Expr         -- e1 e2
```

Intérpretes para lenguajes funcionales

Es teóricamente posible programar usando el cálculo-λ sin extensiones, pero vamos a considerar algunas extensiones para que se asemeje un poco más a un lenguaje realista.

```
data Expr =
          ExprVar Id                -- x
        | ExprLam Id Expr           -- λx.e
        | ExprApp Expr Expr         -- e1 e2
        | ExprLet Id Expr Expr      -- let x = e1 in e2
        | ExprConstNum Int          -- n
        | ExprConstBool Bool        -- b
        | ExprAdd Expr Expr         -- e1 + e2
        | ExprIf Expr Expr Expr -- if e1 then e2 else e3
```

Intérpretes para lenguajes funcionales

¿Cuál va a ser el significado de una expresión como $\lambda x.(x + x)$?

Como en los lenguajes funcionales **las funciones son valores**, necesitamos extender el tipo de los valores.

Una primera aproximación es representar una función con su **código fuente**.

```
data Val = VN Int
         | VB Bool
         | VFunction Id Expr
```

De este modo el significado de $\lambda x.(x + x)$ sería:

```
VFunction "x" (ExprAdd (ExprVar "x") (ExprVar "x"))
```

Intérpretes para lenguajes funcionales

Ejercicio. Definir `eval06Dyn :: Expr -> Env Val -> Val`.

Ejercicio. Evaluar el siguiente programa:

```
let suma = λx.λy.(x + y) in
let f = suma 5 in
let x = 0 in
  f 3
```

Intérpretes para lenguajes funcionales

- El problema es que la variable `f` queda ligada al siguiente valor:

```
VFunction "y" (ExprAdd (ExprVar "x") (ExprVar "y"))
```

- Dicho valor tiene a `x` como variable **libre**.
- Esta técnica se llama **scope dinámico** y así funcionaban los lenguajes funcionales arcaicos (LISP 1.0, Emacs Lisp).
- Es más un error de diseño que una "técnica".
- En esa época ya se lo consideraba un problema (el *funarg problem*).

Intérpretes para lenguajes funcionales

- La técnica correcta para evaluar funciones que pueden contener variables libres es la que se conoce como **scope léxico**.
- La expresión $\lambda y.(x + y)$ evaluada en un entorno E da como resultado un valor de la siguiente forma, que incluye el código fuente de la función y también el entorno E:

```
VClosure "y" (ExprAdd (ExprVar "x") (ExprVar "y")) E
```

- Estos valores comúnmente se conocen como "clausuras léxicas".

```
data Val = VN Int
         | VB Bool
         | VClosure Id Expr (Env Val)
             -- clausura de (λx.e) en el entorno E
```

Intérpretes para lenguajes funcionales

Ejercicio. Definir `eval07Lex :: Expr -> Env Val -> Val` usando *scope* léxico.

Ejercicio. Evaluar el siguiente programa con el nuevo intérprete:

```
let suma = λx.λy.(x + y) in
let f = suma 5 in
let x = 0 in
  f 3
```

```
data Expr =
         ExprVar Id                -- x
       | ExprLam Id Expr           -- λx.e
       | ExprApp Expr Expr         -- e1 e2
       | ExprConstNum Int          -- n
       | ExprConstBool Bool        -- b
       | ExprAdd Expr Expr         -- e1 + e2
       | ExprLet Id Expr Expr      -- let x = e1 in e2
       | ExprIf Expr Expr Expr -- if e1 then e2 else e3
data Val = VN Int | VB Bool | VClosure Id Expr (Env Val)
```

Call-by-value vs. call-by-name

La evaluación de la aplicación $e_1\,e_2$ y del let se podrían definir de varias maneras.

La estrategia de evaluación que usamos hasta ahora se conoce como **call-by-value**:

- Los argumentos se evalúan antes de evaluar el cuerpo de la función.

$$
\begin{array}{l}
[\![e_1\,e_2]\!]E \;=\; \\
\quad \texttt{let v1 = } [\![e_1]\!]E \texttt{ in} \\
\quad\quad \texttt{case v1 of} \\
\quad\quad\quad \texttt{VClosure x } e_{body} \texttt{ E' ->} \\
\quad\quad\quad\quad \texttt{let v2 = } [\![e_2]\!]E \texttt{ in} \\
\quad\quad\quad\quad\quad [\![e_{body}]\!]\ E'[x := v2] \\
\quad\quad\quad \texttt{\_ -> error}
\end{array}
$$

- En una expresión `(let x = e1 in e2)`, se evalúa `e1` antes de evaluar el cuerpo.

Call-by-value vs. call-by-name

Otra estrategia de evaluación posible se conoce como **call-by-name**:

- El parámetro formal se liga al argumento real no evaluado, y se lo evalúa cada vez que se lo necesita.
- En una expresión `(let x = e1 in e2)`, se evalúa `e2` directamente, manteniendo la variable `x` ligada a una copia sin evaluar de `e1`.
- En call-by-name los entornos no asocian identificadores a valores, sino identificadores a **expresiones aún no evaluadas, clausuradas en un entorno**.

Call-by-value vs. call-by-name

Ejercicio. Definir un intérprete para un lenguaje funcional call-by-name.

```
eval08ByName :: Expr -> Env Thunk -> Val

data Thunk = TT Expr (Env Thunk)

data Expr =
          ExprVar Id                -- x
        | ExprLam Id Expr           -- λx.e
        | ExprApp Expr Expr         -- e1 e2
        | ExprLet Id Expr Expr      -- let x = e1 in e2
        | ExprConstNum Int          -- n
        | ExprConstBool Bool        -- b
        | ExprAdd Expr Expr         -- e1 + e2
        | ExprIf Expr Expr Expr -- if e1 then e2 else e3

data Val = VN Int | VB Bool | VClosure Id Expr (Env Thunk)
```

Call-by-value vs. call-by-name

- **Detalle importante:** en los intérpretes con entornos como los recién estudiados, la estrategia de evaluación depende de la estrategia del lenguaje de implementación.
- El intérprete *call-by-value* definido arriba **no** es verdaderamente *call-by-value* si se lo ejecuta en un lenguaje como Haskell que utiliza la estrategia *call-by-need* ("*lazy*").
- Hay maneras de definir las estrategias de evaluación de manera interna, es decir, sin depender de la estrategia de evaluación usada por el lenguaje de implementación.
- Usar continuaciones es una manera de asegurarse de que el orden de evaluación no dependa de la estrategia usada por el lenguaje subyacente.
- Otra alternativa es definir la semántica por otros medios (p.ej. operacional, denotacional) que no vamos a tratar en esta materia.

Lenguajes con características funcionales e imperativas

La mayoría de los lenguajes modernos combinan características funcionales e imperativas: admiten funciones de "alto orden" y variables mutables.

Nota: desde el punto de vista de la semántica dinámica no hay grandes diferencias entre una clausura léxica y un objeto.

- Las clausuras son objetos que responden a un único mensaje: aplicar.

Objects are a poor man's closures.
Closures are a poor man's objects.

Lenguajes con características funcionales e imperativas

Podemos combinar el intérprete imperativo *store-passing* y el intérprete funcional call-by-value con *scope* léxico:

Ejercicio. Definir la función

```
eval09LexMem :: Expr -> Env Addr -> Memory Val
                                 -> (Val, Memory Val)
```

que combine las características anteriores.
(Este ejercicio queda literalmente como ejercicio.)

```
data Expr =
          ExprVar Id                -- x
        | ExprAssign Id Expr        -- x := e
        | ExprLam Id Expr           -- λx.e
        | ExprApp Expr Expr         -- e1 e2
        | ExprLet Id Expr Expr      -- let x = e1 in e2
        | ExprConstNum Int          -- n
        | ExprConstBool Bool        -- b
        | ExprAdd Expr Expr         -- e1 + e2
        | ExprIf Expr Expr Expr -- if e1 then e2 else e3
data Val = VN Int | VB Bool | VClosure Id Expr (Env Val)
```

Expansión o "desazucarado"

Muchas construcciones de lenguajes de programación se pueden expresar en términos de otras construcciones de ese mismo lenguaje. Esta es una primera manifestación de la **compilación**.

Algunos lenguajes, y en especial los lenguajes funcionales, "desazucaran" (*desugar*) el programa de entrada y trabajan con un programa equivalente escrito en un subconjunto reducido del lenguaje.

Expansión del for

Ejemplo. Expansión de la construcción `for i = a to b`.

Código fuente	Expansión

`for i = <expr1> to <expr2>`
`    <block>`
`end`
	`i := <expr1>`
	`_limite := <expr2>`
	`while i <= _limite`
	`    <block>`
	`    i := i + 1`
	`end`

Sutilezas importantes:

- ¿Por qué es necesario guardar el valor de `<expr2>` en `_limite`?
- El nombre de la variable auxiliar `_limite` debe estar elegido de tal manera que no haya conflicto con nombres de variable escritos por el usuario, ni con otras variables auxiliares (por ejemplo, las que introduzca un for anidado).

Funciones recursivas

Las funciones recursivas son una construcción fundamental en los lenguajes de programación. Los intérpretes que hicimos hasta el momento no alcanzan para interpretar un lenguaje con definiciones recursivas.

Ejercicio. Evaluar el siguiente programa con la función `evalLex` (call-by-value con *scope* léxico) y comprobar que el `let` que definimos no admite definiciones recursivas:

```
let f = λx.f (x + 1) in
  f 1
```

Funciones recursivas

Una manera de implementar funciones recursivas si disponemos de memoria es con la siguiente expansión:

Código fuente	Expansión

`letrec`
f_1 `=` `<expr`$_1$`>`
f_2 `=` `<expr`$_2$`>`
`...`
f_n `=` `<expr`$_n$`>`
`in`
`<cuerpo>`
	`let` f_1 `=` `nil` `in`
	`let` f_2 `=` `nil` `in`
	`...`
	`let` f_n `=` `nil` `in`
	f_1 `:=` `<expr`$_1$`>`
	f_2 `:=` `<expr`$_2$`>`
	`...`
	f_n `:=` `<expr`$_n$`>`
	`<cuerpo>`

Interpretación de tuplas

Ejercicio. Extender el intérprete:

```
evalLexMem :: Expr -> Env Addr -> Memory Val
                               -> (Val, Memory Val)
```

para las siguientes construcciones:

```
data Expr = ...
          | ExprTuple [Expr]        -- ⟨e1, e2 ..., en⟩
          | ExprProj Int Expr       -- πk(e)

data Val = ...
         | VT [Val]
```

Expansión de los tipos de datos inductivos

Los tipos de datos inductivos se pueden "desazucarar" a números y tuplas.

Código fuente	Expansión
`data Color = Blanco` `  | Negro`	`1` `2`
`data Forma = Circulo Color Int` `  | Cuadrado Color Int Int`	`[1, c, n]` `[2, c, n1, n2]`

`data T a = Nil`
`  | Leaf a`
`  | Bin a (T a) (T a)`
	`[1]`
	`[2, a]`
	`[3, a, t1, t2]`

Ejemplo.

```
Bin (Cuadrado Negro 10 20) Nil (Leaf (Circulo Blanco 30))
```

Se desazucara así:

```
[3, [2, 2, 10, 20], [1], [2, [1, 1, 30]]]
```

Expansión de los tipos de datos inductivos

El *pattern matching* se puede desazucarar (de manera precaria) usando `ifs` y las proyecciones π_k de las tuplas.

Ejemplo.

```
letrec nroNodos = λarbol.
          case arbol of
            Nil          -> 0
            Leaf a       -> 1
            Bin a t1 t2 -> 1 + nroNodos t1
                             + nroNodos t2
  in nroNodos
```

Se desazucara así:

```
letrec nroNodos = λarbol.
          if π1(arbol) == 0
           then 0
           else if π1(arbol) == 1
                 then 1
                 else 1 + nroNodos π3(arbol)
                        + nroNodos π4(arbol)
  in nroNodos
```

Evaluación call-by-need

Algunos lenguajes (como Haskell) usan la estrategia de evaluación que se conoce como call-by-need. Call-by-need combina las ventajas de call-by-name y call-by-value: los argumentos se evalúan sólo cuando es necesario, pero el resultado de la evaluación se comparte entre las distintas copias.

En call-by-need hay dos tipos de valores:

- Valores ya evaluados (números, booleanos, clausuras, etc.).
- Valores que todavía no fueron evaluados (*thunks*).

La evaluación call-by-need obedece a las siguientes reglas:

- Las direcciones de memoria pueden almacenar valores ya evaluados o *thunks*.
- Cuando se evalúa una expresión, el resultado final que se devuelve no es un *thunk* (si es un *thunk*, hay que seguir evaluando).

Evaluación call-by-need

Ejercicio. Definir un intérprete call-by-need para el lenguaje que se da a continuación.

```
evalNeed :: Expr -> Env Addr -> Memory Val
                             -> (Val, Memory Val)

data Expr =
          ExprVar Id                -- x
        | ExprLam Id Expr           -- λx.e
        | ExprApp Expr Expr         -- e1 e2
data Val = VN Int | VB Bool | VClosure Id Expr (Env Addr)
          | VThunk Expr (Env Addr)
```

Observar que la memoria es un componente indispensable para la evaluación call-by-need, pese a que este lenguaje no cuenta con variables mutables.