Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación

Nota de clase 3: Ambientes de evaluación y alcance

Karla Ramírez Pulido
karla@ciencias.unam.mx

Manuel Soto Romero
manu@ciencias.unam.mx

13 de octubre 2017
Semestre 2018-1

Descripción

Los lenguajes de programación usados en la industria (como C y C++) y en la academia (como Haskell y Racket) usan funciones (subrutinas, métodos, etcétera) para facilitar la resolución de problemas, sin embargo, no tienen el misma comportamiento en todos los lenguajes. En esta nota se revisarán los tipos de funciones que existen, hablando de lenguajes de programación y los retos que conlleva su implementación.

Índice general

Índice de códigos

Índice de figuras

4.1 El lenguaje FWAE

En estas notas se extiende el lenguaje `WAE` estudiado en las Notas de Laboratorio 3, agregando funciones para discutir los distintos retos en su implementación. La gramática en notación EBNF de este lenguaje se muestra en la Figura 1.

```
<expr> ::= <id>
         | <num>
         | {<binop> <expr> <expr>}
         | {with {<id> <expr>} <expr>}
         | {fun {<id>} <expr>}
         | {<expr> <expr>}

<id> := a |...|z|A|...|Z|aa|...|zz|...
        (Cualquier conmbinación de carácteres)

<num> := ...|-2|-1|0|1|2|...

<binop> := +|-|*|/
```

Figura 1: *Gramática del lenguaje* `FWAE`

Ejemplo 4.1 Algunos ejemplos de expresiones dentro del lenguaje `FWAE`.

- Identificadores:

  ```
  foo
  ```

- Números:

  ```
  1729
  ```

- Operaciones binarias:

  ```
  {+ 17 {- 29 {* 18 {/ 35 40}}}}
  ```

- Asignaciones locales:

  ```
  {with {a 2}
     {+ a 3}}
  ```

- Funciones:

  ```
  {fun {y} {+ 10 y}}
  ```

- Aplicación de funciones:

  ```
  {{fun {y} {+ 10 y}} 2}
  ```

- Una expresión más compleja

```
{with {x 3}
   {with {f {fun {y} {+ x y}}}
      {with {x 5}
         {f 4}}}}
```

•

Clasificación de funciones

Cuando se habla funciones en un lenguaje de programación, existen algunas diferencias en cuanto a su comportamiento, dependiendo del lenguaje usado. Por ejemplo, las siguientes expresiones son validas en un lenguaje como Haskell:

```
duplica x = 2 * x

duplicados = map double [1,2,3,4]
```

Código 1: *Uso de funciones en Haskell*

La función `map` recibe otra función, en este caso la función `duplica`, como parámetro. Haskell a su vez provee mecanismos para regresar funciones como resultado, e incluso almacenarlas como valor. En otros lenguajes como C esto no es posible, ya que las funciones sólo pueden aplicarse a valores, pero jamás son recibidas como parámetro, por ejemplo.

Las funciones de un lenguaje de programación, se pueden clasificar como sigue[2]:

- ***Funciones de primer orden***. *Las funciones no son consideradas valores en el lenguaje.*
- ***Funciones de orden superior***. *Las funciones pueden regresar otras funciones como valor.*
- ***Funciones de primera clase***. *Las funciones son tratadas como cualquier otro valor del lenguaje. Pueden pasarse como parámetro a otras funciones, regresarse como valor e incluso almacenarse.*

Los intérpretes que se construirán a lo largo de esta nota, usarán funciones de primera clase.

Análisis sintáctico de FWAE

Para construir el Árbol de Sintaxis Abstracta (ASA) de las expresiones de `FWAE` se utiliza el `TDA FWAE` definido en el Código 2.

```
;; TDA para construir el árbol de sintaxis abstracta de
;; las expresiones de FWAE
(define-type FWAE
   [idS (i symbol?)]
   [numS (n number?)]
   [binopS (f procedure?) (izq FWAE?) (der FWAE?)]
   [withS (id symbol?) (val FWAE?) (body FWAE?)]
   [funS (param symbol?) (body FWAE?)]
   [appS (fun FWAE?) (arg FWAE?)])
```

Código 2: *Tipo de dato abstracto* `FWAE`

De esta forma, en análisis sintáctico se realiza de forma parecida al descrito en la Nota de Laboratorio 3 y se muestra en el Código 3.

```
;; parse: s-expression -> FWAE
(define (parse sexp)
   (match sexp
      [(? symbol?) (idS sexp)]
      [(? number?) (numS sexp)]
      [(list 'with (list id val) body)
         (withS id (parse val) (parse body))]
      [(list 'fun (list param) body) (funS param (parse body))]
      [(list op l r) (binopS (elige op) (parse l) (parse r))]
      [else (appS (parse (car sexp)) (parse (cadr sexp)))]))
```

Código 3: *Función* `parse` *que realiza el análisis sintáctico de* `FWAE`

4.2 Azúcar sintáctica

Antes de implementar el analizador semántico `FWAE`, se realiza una posible etapa intermedia de transformación que realizan algunos lenguaje de programación modernos, la cual consiste en quitar el azúcar sintáctica de las expresiones del lenguaje[2]. Existen otros tipos de transformaciones para realizar optimizaciones, por ejemplo, el azúcar sintáctica es un tipo de sintaxis especial que facilita la escritura de algunas expresiones en el lenguaje, haciendo éstas más amigables o "dulces" para el usuario. Por ejemplo, en Racket, cuando se escribe:

```
(let* ([a 2] [b a])
   b)
```

Esto es en realidad una versión simplificada que facilita la escritura de:

```
(let ([a 2])
   (let ([b a])
      b)))
```

De esta forma, Racket realiza una etapa de transformación al código para eliminar el azúcar sintáctica de expresiones como la anterior. En el TDA para FWAE, también se tienen expresiones endulzadas, como las asignaciones locales (with). Por ejemplo:

```
{with {a 2}
   {+ a a}}
```

la cual es una versión endulzada de:

```
{{fun {a} {+ a a}} 2}
```

De esta forma, las asignaciones locales, son en realidad aplicaciones de función endulzadas. Se necesita, entonces, de un mecanismo que elimine este tipo de endulzamiento, para ello, se define en el Código 4 un TDA para FAE que no incluye expresiones with, pues éstas ya no son necesarias.

```
;; TDA para construir el árbol de sintaxis abstracta de
;; las expresiones de FAE, una versión sin azúcar sintáctica
;; de FWAE
(define-type FAE
   [id (i symbol?)]
   [num (n number?)]
   [binop (f procedure?) (izq FAE?) (der FAE?)]
   [fun (param symbol?) (body FAE?)]
   [app (fun FAE?) (arg FAE?)])
```

Código 4: *Tipo de dato abstracto* FAE

Para realizar el mapeo entre expresiones de FWAE a expresiones de FAE (quitando el azúcar sintáctica), se define la función desugar en el siguiente Código 5.

```
;; desugar: FWAE -> FAE
(define (desugar sexp)
   (match sexp
      [(idS i) (id i)]
      [(numS n) (num n)]
      [(binopS f izq der) (binop f (desugar izq) (desugar der))]
      [(withS id val body)
         (app (fun id (desugar body)) (desugar val))]
      [(funS param body) (fun param (desugar body))]
      [(appS fun arg) (app (desugar fun) (desugar arg))]))
```

Código 5: *Función* desugar *que elimina el azúcar sintáctica de* FWAE

4.3 Análisis semántico de FWAE

Ahora que se tiene una versión desendulzada de FWAE, el análisis semántico se realiza sobre las expresiones de tipo FAE, lo cual simplifica su implementación pues no se tienen casos distintos para las asignaciones locales y aplicación de funciones (el analizador semántico se muestra en el Código 6). Es importante destacar que este intérprete, al no regresar únicamente números, devuelve ahora expresiones de tipo FAE para evitar futuros errores de tipo.

```
;; interp: FAE -> FAE
(define (interp expr)
   (match expr
      [(id i) (error 'interp "Free id")]
      [(num n) expr]
      [(binop f izq der)
         (num (f (num-n (interp izq)) (num-n (interp der))))]
      [(fun param body) expr]
      [(app fun arg)
         (interp
            (subst
               (fun-body fun)
               (fun-param fun)
               (interp arg)))]))
```

Código 6: *Función* interp *(versión 1)*

Como puede apreciarse, el procedimiento subst también requiere de algunas modificaciones para procesar funciones su aplicación. Éstas se muestran en el Código 7.

```
;; subst: FAE symbol FAE -> FAE
(define (subst expr sub-id val)
   (match expr
      [(id i) (if (symbol=? i sub-id) val expr)]
      [(num n) expr]
      [(binop f izq der)
         (binop f (subst izq sub-id val) (subst der sub-id val))]
      [(fun param body)
         (if (symbol=? param sub-id)
             (fun param body)
             (fun param (subst body sub-id val)))]
      [(app fun arg)
         (app (subst fun sub-id val) (subst arg sub-id val))]))
```

Código 7: *Función* subst

Ejemplo 4.2 Se muestra el resultado de ejecución de cada función sobre una expresión para mostrar los distintos tipos de análisis implementados hasta este momento. Se omiten pasos intermedios que el lector puede seguir a partir de los códigos mostrados anteriormente.

- Entrada: Una expresión en sintaxis abstracta.

```
{with {x 3}
   {with {f {fun {y} {+ x y}}}
      {with {x 5}
         {f 4}}}}
```

- Análisis léxico: Separa la expresión en lexemas.

```
'{with {x 3}
   {with {f {fun {y} {+ x y}}}
      {with {x 5}
         {f 4}}}}
```

- Análisis sintáctico: Construye el árbol de sintaxis abstracta.

```
(parse '{with {x 3} {with {f {fun {y} {+ x y}}} {with {x 5} {f 4}}}})

(withS
   'x
   (numS 3)
   (withS
      'f
      (funS 'y (binopS #<procedure:+> (idS 'x) (idS 'y)))
      (withS 'x (numS 5) (appS (idS 'f) (numS 4)))))
```

- Eliminar azúcar sintáctica: Convierte una expresión `FWAE` a otra de tipo `FAE`

```
(desugar
   (withS
      'x
      (numS 3)
      (withS
         'f
         (funS 'y (binopS #<procedure:+> (idS 'x) (idS 'y)))
         (withS 'x (numS 5) (appS (idS 'f) (numS 4))))))

(app
   (fun
      'x
      (app
         (fun 'f (app (fun 'x (app (id 'f) (num 4))) (num 5)))
         (fun 'y (binop #<procedure:+> (id 'x) (id 'y)))))
   (num 3))
```

- Análisis semántico: Evalúa el Árbol de Sintaxis Abstracta (ASA) sin azúcar sintáctica

```
(interp
   (app
      (fun
         'x
         (app
            (fun 'f (app (fun 'x (app (id 'f) (num 4))) (num 5)))
            (fun 'y (binop #<procedure:+> (id 'x) (id 'y)))))
      (num 3)))

(num 7)
```

•

Ambientes de evaluación

En el Ejemplo 4.2 se puede apreciar el algoritmo de sustitución textual el cual puede resultar complejo, pues por cada expresión `with` que aparece se llama al algoritmo de sustitución (en este caso 3 veces). Si el programa tuviera un tamaño n (el número de nodos del árbol de sintaxis abstracta), entonces cada sustitución recorrerá el resto del programa al menos una vez, haciendo que su complejidad se vuelva cuadrática, es decir, $O(n^2)$.

Para bajar este orden, se propone el uso de *ambientes* de evaluación implementados con alguna estructura de datos en la cual las búsquedas sean de al menos complejidad lineal, es decir, $O(n)$. Para los fines de estas notas, se usará un ambiente con comportamiento de pila pues su complejidad es lineal. Para representar los ambientes se define en el Código 8 el TDA `Env`.

```
;; TDA que representa el ambiente de evaluación.
(define-type Env
   [mtSub]
   [aSub (name symbol?) (value FAE?) (env Env?)])
```

Código 8: *Tipo de dato abstracto* `Env`

De esta forma, la sustitución de identificadores se realiza buscando el valor correspondiente en el ambiente. De forma gráfica podríamos pensar en el ambiente como una tabla donde la primera columna almacena los identificadores correspondientes y la segunda el valor asociado, la Figura 2 muestra esta representación.

De la figura anterior, se puede observar que es la representación gráfica del siguiente código:

```
{with {a 2}
   {with {b 3}
      {with {c 4}
         {+ a {+ b c}}}}}
```

name	value
'c	(num 4)
'b	(num 3)
'a	(num 2)

Figura 2: *Ambiente de evaluación de forma gráfica*

Nótese que los identificadores ingresan en forma de pila conforme se van leyendo en la expresión, es decir, como aparecen en el código del programador.

Para buscar un identificador el ambiente, se presenta el Código 9 el cual muestra la implementación de la función `lookup` que va comparando el valor deseado con cada identificador en el ambiente. La búsqueda se realiza recursivamente.

```
;; lookup: symbol Env -> FAE
(define (lookup id env)
   (match env
      [(mtSub) (error 'lookup "Free Id")]
      [(aSub name value rest-env)
         (if (symbol=? name id)
             value
             (lookup id rest-env))]))
```

Código 9: *Función* `lookup`

De esta forma, la nueva versión del analizador semántico se muestra en el Código 10, esta nueva versión recibe además de la expresión, un ambiente de evaluación.

```
;; interp: FAE Env -> FAE
(define (interp expr env)
   (match expr
      [(id i) (lookup i env)]
      [(num n) expr]
      [(binop f izq der)
         (num (f (num-n (interp izq env)) (num-n (interp der env))))]
      [(fun param body) expr]
      [(app fun arg)
         (let ([fun-val (interp fun env)])
            (interp
               (fun-body fun-val)
               (aSub (fun-param fun-val) (interp arg env) env)))]))
```

Código 10: *Función* `interp` *(versión 2)*

Las aplicaciones de función deben interpretar el cuerpo de la función correspondiente en el ambiente que se construye ingresando el parámetro formal de la función asociado al argumento que recibe la aplicación de función. De esta forma, la expresión del Ejemplo 4.2[1]:

[1]Recordar que un `with` es azúcar sintáctica de una aplicación de función.

```
{with {x 3}
   {with {f {fun {y} {+ x y}}}
      {with {x 5}
         {f 4}}}}
```

se evalúa en el siguiente ambiente:

name	value
'x	(num 5)
'f	(fun 'y (binop + (id 'x) (id 'y)))
'x	(num 3)

¿Qué valor de `'x` debería tomar la función `{fun {y} {+ x y}}` el de arriba o el de abajo?

4.4 Alcance

Después de que el lector haya ejecutado el programa anterior, se observa que el intérprete regresa un valor de `(num 9)` siendo que en el Ejemplo 4.2 se obtuvo un valor de `(num 7)` ¿cuál es entonces el resultado correcto?

Esto tiene que ver con el concepto de *alcance*. La intuición nos dice que en la expresión:

```
{with {x 3}
   {with {f {fun {y} {+ x y}}}
      {with {x 5}
         {f 4}}}}
```

la `{fun {y} {+ x y}}` debe tomar el valor de `{x 3}` pues es dónde fue definida la función. Sin embargo, también se tiene la posibilidad de tomar el valor `{x 5}` pues es el valor actual de `x` cuando se manda a llamar la función[2]. Estas dos posibles tipos de alcance tienen nombre[2]:

- ***Alcance dinámico.*** *El valor de los identificadores se toma de la región dónde éstos son llamados.*
- ***Alcance estático.*** *El valor de los identificadores se toma revisando todo el programa o expresión.*

De esta forma, la última versión de `interp` utiliza alcance dinámico al tomar valores dentro de la región donde es llamada la función, con lo cual `x = 5` y por lo tanto `{f 4} = {+ 5 4} = 9`. Por otro lado, la primera versión de `interp` usa alcance estático pues tomaba valores revisando todo el programa desde el inicio, donde se encontraba con que `x = 3`, por lo tanto `{f 4} = {+ 3 4} = 7`.

Ambos tipos de alcance son correctos y dependerán de la visión que tenga el diseñador o los diseñadores del lenguaje, para elegir alguno de estos, sin embargo, es común usar alcance estático pues es lo que la intuición del usuario espera.

La primera versión del analizador semántico (función `interp`) implementada alcance estático pues realizaba la sustitución antes que otra cosa se ejecutara, por ejemplo, la subexpresión

```
{with {x 3}
   {with {f {fun {y} {+ x y}}}
      ...}}
```

era sustituida al inicio por

```
{with {f {fun {y} {+ 3 y}}}
   ...}}
```

Necesitamos entonces, que el intérprete basado en ambientes, en lugar de evaluar funciones a sí mismas, cargue el ambiente donde éstas fueron definidas. Este tipo de estructura recibe el nombre de *cerradura* (del inglés *closure*) y almacena[2]:

- Los parámetros formales de la función.
- El cuerpo de la función.
- El ambiente dónde fue definida la función.

Para representar las cerraduras, se define el siguiente tipo de dato, que es el valor de regreso del nuevo intérprete:

```
;; TDA para representar los resultados del intérprete
;; Permite definir cerraduras de función.
(define-type FAE-Value
   [numV (n number?)]
   [closureV (param symbol?) (body FAE?) (env Env?)])
```

Código 11: *TDA* `FAE-Value` *para representar las cerraduras*

De esta forma al evaluar aplicaciones de función, el valor de los identificadores se debe buscar en el ambiente de la cerradura y no en el ambiente actual. El Código 12 muestra la nueva versión de `interp`.

```
;; interp: FAE Env -> FAE-Value
(define (interp expr env)
   (match expr
      [(id i) (lookup i env)]
      [(num n) (numV n)]
      [(binop f izq der)
         (numV
            (f
               (numV-n (interp izq env))
               (numV-n (interp der env))))]
      [(fun param body) (closureV param body env)]
      [(app fun arg)
         (let ([fun-val (interp fun env)])
```

```
            (interp
               (closureV-body fun-val)
                  (aSub
                     (closureV-param fun-val)
                     (interp arg env)
                     (closureV-env fun-val))))]))
```

Código 12: *Función* `interp` *(versión 3)*

También debe modificarse la definición de `Env` pues ahora debe almacenar valores de tipo `FAE-Value`.

```
;; TDA que representa el ambiente de evaluación.
(define-type Env
   [mtSub]
   [aSub (name symbol?) (value FAE-Value?) (env Env?)])
```

Código 13: *Tipo de datos abstracto* `Env` *(versión 2)*

4.5 Referencias

[1] Rodrigo Ruiz Murgía, *Manual de prácticas para la asignatura de Lenguajes de Programación*, Reporte de actividad docente, Facultad de Ciencias, 2016.

[2] Shriram Krishnamurthi, *Programming Languages: Application and Interpretation*, Primera edición, Brown University, 2007.

[3] Favio E. Miranda Perea, Elisa Viso Gurovich, *Matemáticas Discretas,* Primera edición, Las prensas de Ciencias, 2009.

[4] Keith D. Cooper, Linda Torczon, *Engineering a Compiler,* Segunda edición, Morgan Kaufman, 2012.

Nota: Esta es una versión preliminar de las notas de laboratorio y se encuentran en constante actualización por lo que agradecemos que en caso de detectar algún error o si se desea hacer alguna observación se envíe un correo electrónico a las direcciones `manu@ciencias.unam.mx` y `karla@ciencias.unam.mx` con el asunto `Nota L4: Lenguajes de Programación`.